UiO : Department of Informatics
University of Oslo

Implementing Garbage Collection for Active Objects on Top of Erlang

Sigmund Hansen
master thesis autumn 2014

Implementing Garbage Collection for Active Objects on Top of Erlang

Sigmund Hansen

December 15, 2014

Abstract

ABS is a language for modeling and simulating distributed systems. ABS has been in development at the University of Oslo for a number of years and has been the technical underpinning for a number of national and EU-wide research projects such as HATS and Envisage.

The key characteristic of ABS is a semantics of *active objects* encapsulating parallel behavior in a safe way. Objects communicate via asynchronous method calls and future variables.

Modern programming languages typically relieve programmers of the burden of manually managing memory. Errors due to memory leaks and dangling pointers has been a headache in low-level languages like C. ABS has an Erlang back end that has been developed as part of a master's thesis by Georg Göri at the Graz University of Technology. Although Erlang is a garbage collected language, it cannot collect idle processes.

In this thesis, a garbage collector that stops processes that represent active objects, future variables and schedulers in the Erlang back end, has been developed. The thesis discusses measures taken to ensure correct collection of these processes, as well as strategies employed to balance completeness and speed.

Contents

List of Figures

List of Tables

List of Listings

Preface

The feeling is bittersweet as the end of my studies draws near, like it so often is when one thing ends and another begins.

First and foremost I would like to thank my supervisors: Rudolf Schlatte for suggesting the project and his technical help throughout the project; and Ingrid Chieh Yu who helped find me a new master's project, all her help improving my writing and for the years teaching algorithms and data structures.

Thanks to Lars Tveito, who looked over my thesis at the last minute, finding several cases missing words and silly typos.

I would also like to thank all the people I've had the pleasure of working with here at the University of Oslo. Particularly Roger Antonsen, for his unbreakable spirit and incredible dedication to the department and students, especially with establishing the Department's hackerspace, *Åpen sone for eksperimentell informatikk*. Thanks go out to the student council for the good times we've shared, and specifically José Luis Rojas who pushed me to apply for a teaching assistant position, a job that has meant a lot to me and helped me grow as a person, all the students and professors on the 8th floor, to my family and in particular my brother, who is probably to blame for my interest in computing, and all my teachers over the years.

Last, but not least, thank you, Tsz Yan Tong, for your love and support.

Chapter 1

Introduction

The overall goal of this thesis is to add garbage collection to the ABS language's Erlang back end. ABS is a language for creating executable models of distributed systems using active objects. Adding garbage collection should allow larger systems, that would have exhausted resources without it, to be modeled and tested.

Modern computers typically have multiple processing units, which come in the form of multiple cores on a single chip, multiple CPUs, virtual cores and GPUs capable of general purpose processing. They typically run many processes at the same time that may communicate with one another as part of a larger system.

In addition to modern hardware, virtualization has made large scale clusters much more available; through the click of a button a new virtual computer can be brought online via one of many Infrastructure or Platform as a Service providers. Software systems have increased greatly in scale and are often used for years or decades. Ensuring that these systems are maintainable and adaptable throughout their long lifetimes, is challenging without appropriate prior modeling.

The HATS project (*Highly Adaptable and Trustworthy Software using Formal Methods*) was established to research formal modeling of distributed systems. One of the fruits of this project is a modeling language called ABS specifically targeting distributed systems, which is now being developed in the Envisage project (*Engineering Virtualized Services*).

Memory is a finite resource that programs must avoid exhausting. Low level languages, like C, usually leaves this up to the programmer. While high level languages, like Java, often include automatic memory management, known as garbage collection. ABS can compile to Erlang, a garbage collected language that compiles to byte code. Large parts of the resulting programs, however, are modeled as processes which are not seen as garbage by the built-in collector, unless they stop running. Thus there is need for a special purpose garbage collector for the compiled ABS models to enable freeing memory in these models.

1.1 Goals

The garbage collector should be:

- Fast - It should not invoke considerable slowdowns to the system.
- Comprehensive - It should collect all or, at the very least, most identifiable garbage.
- Correct - It should not collect objects that may be needed in the future.

For the collector to be fast, it should do as little work as possible. Collecting nothing, would be the fastest alternative, but obviously would not improve memory usage. On the other hand, to be comprehensive, will require more work to ensure garbage is collected. The fastest alternative that guarantees collecting all garbage, is to skip the analysis and just collect everything. However, this would also collect objects that may be needed in the future, resulting in incorrect collection. In this thesis, I will therefore attempt to strike a balance between speed and completeness without compromising correctness.

1.2 Structure of This Thesis

Chapter 2 introduces the ABS language. In chapter 3, Erlang and the Erlang back end for ABS is described. A general introduction to garbage collection techniques and problems is given in chapter 4. The implemented garbage collector is described in chapter 5 and evaluated in chapter 6. Finally, chapter 7 summarizes the findings and concludes this thesis.

Chapter 2

The ABS Language

ABS is a modeling language for *abstract behavioral specifications*, that targets modeling of concurrent, distributed systems [10]. The syntax is made to resemble Java's, making it easy to learn for developers familiar with Java and similar languages. It uses explicit typing and static type checking similarly to Java. ABS is divided into two levels, a functional level and a concurrent object-oriented level, where the OO level builds on the functional level.

Most specification languages are either design oriented, implementation oriented or foundational languages. ABS is situated somewhere between these three types. Like implementation-oriented languages, "it is designed to be close to the way programmers think, by maintaining a Java-like syntax and a control flow close to an actual implementation". ABS has formally defined semantics like foundational languages. Some implementation details are abstracted away, such as the details of the communication environment and scheduling policies [10]. This allows the modeler to focus on the relationship between classes and the flow of messages between them, as is the focus of design-oriented languages, instead of low-level details.

ABS has an associated tool suite with compilers, a unit testing framework, an Eclipse plug-in, an Emacs mode, debugging and analysis tools. A specification can be compiled to one of multiple back end languages. The Maude back end allows verification while other languages allow simulation.

2.1 Functional Level

A subset of the ABS language is its functional constructs consisting of pure expressions, pure functions and algebraic data types. Pure expressions are free of side effects, meaning they cannot modify the heap, and pure functions consist of pure expressions. Algebraic data types are often featured in languages that use pattern matching, as values are literals suited for pattern matching. Unlike most functional languages, ABS has no higher order functions.

2.1.1 Algebraic Data Types

ABS has algebraic data types, which users may define as a set of one or more constructors. Constructors may take typed arguments, and a constructor with applied parameters is the literal representation of a value of such a type. ABS's algebraic types support polymorphism in the form of type parameters. Type parameters are given in the same form as in Java, enclosing a list of type names in angle brackets. There are several built-in data types such as: `Bool`, `Int`, `Rat`, `String` and `List<A>`. For a full overview of built-in types, their implementations and associated functions, see *The ABS Language Specification* [15, Appendix A].

Two examples of algebraic data types are given in listing 2.1. The first shows a simple enumeration, where none of the constructors take arguments. The second shows an example of a general binary tree, which takes one type parameter for the type of the values. The binary tree type has two constructors, one takes no arguments and represents an empty tree. The other constructor, `Branch`, takes three arguments, a left sub-tree, a value stored within it and a right sub-tree.

Listing 2.1: Examples of algebraic data type definitions.

```
/** Graph coloring enumeration. */
data Color = White | Gray | Black;

/** An algebraic type for general binary trees. */
data Tree<V> = EmptyTree
             | Branch(Tree<V> left, V value, Tree<V> right);
```

Note that some types, like `Int`, are not implemented in ABS, but depend on the back end language, see section 2.3. These types, which are sometimes called basic types, are still algebraic types. For instance `Int` and `String` do not use constructors like the typical algebraic types, but instead have a special syntax for literals. They can still be used in pattern matching, which is covered in the next section.

2.1.2 Pure Expressions

Pure expressions are expressions without side effects. This includes the typical arithmetic, logical and comparison expressions. Variable access including accessing the active object, `this`, or its fields is a kind of pure expression. The use of literal values, that is constructing data of algebraic data types, is a pure expression. Function application is also a pure expression as function bodies consist of a pure expression. Note that functions in this context does not include methods, also known as member functions, in the object-oriented part of the language. Any pure expression can be parenthesized.

In addition to these simple cases, ABS features a few other kinds of expressions often found in functional programming languages. The if-then-else expression, or simply `if` expression, is one, similar to `if`

statements, but the `else` branch is not optional. Like all expressions the `if` expression returns a value. It works by conditionally evaluating one of the two expressions. `if a > b then a else b` is a simple example that shows how to select the largest of two values.

ABS features pattern matching with its `case` expressions. This allows more fine-grained choices of expressions depending on the value of an expression. There are four kinds of patterns: literal values like `123`, variables like `x` which become bound if they are not, or whose value is matched similar to literals if they are already bound, `_` matches anything and ignores the value, and lastly, constructor patterns which use an algebraic constructor where each parameter is matched with a pattern as well, like `Branch(_, x, _)`.

Listing 2.2: An example showing different kinds of patterns supported by ABS's case expression.

```
/** Inserts an element into a binary search tree. */
def Tree<V> insert<V>(V value, Tree<V> tree) =
    case tree {
        EmptyTree => Branch(EmptyTree, value, EmptyTree);
        Branch(left, v, right) =>
            case Pair(value < v, value > v) {
                Pair(False, False) => tree;
                Pair(True, _) =>
                    Branch(insert(value, left), v, right);
                Pair(_, True) =>
                    Branch(left, v, insert(value, right));
            };
    };
```

The example in listing 2.2 shows several variants of constructor patterns. `Pair<A,B>` is a built-in type for two-tuples, and is used as an alternative to nested `if` expressions. In the two last patterns, we ignore the other value in the pair, although we could replace the underscores with `False`. The outer `case` expression shows a slightly more complex pattern: `Branch(left, v, right)`, which will bind values to the three variables for use in the following expression. The example also happens to show the syntax of function definitions, and recursive application of a function.

Lastly we have the `let` expression, which binds the value of a pure expression to a named constant for use in another pure expression. This avoids evaluating an expression multiple times if it is needed in different parts of the expression. This is particularly useful for calling functions that may operate on an entire data structure, and using the result in an `if` expression.

Listing 2.3: Example of let in a recursive function.

```
/** Gets the largest number in a list of natural numbers. */
def Int max(List<Int> l) =
  case l {
    Nil => 0;
    Cons(a, tail) => let (Int b) = max(tail) in
                         if a > b then a else b;
  };
```

2.2 Concurrent Object Level

The functional level is extended by an imperative object-oriented level with side effects. Objects are separated into *concurrent object groups*, abbreviated COGs, where activity in objects within the same group are handled sequentially. Method calls between COGs are done by asynchronous message passing.

2.2.1 Interfaces

ABS uses interfaces for object types. Interfaces have a name that is used as type names. They consist of a set of method declarations, without method definitions. A class that implements to an interface, must have implementations of all the methods of the interface. Because interfaces are the types used for objects, any method that should be accessible from outside the object, must be declared in one of the interfaces that its class implements. Therefore ABS specifications typically have a somewhat larger number of interfaces than traditional systems.

The fact that interfaces are used as types instead of classes also enforces data encapsulation. Interfaces cannot have member variables, therefore any access to members must go through a method defined in an interface.

One limitation to the interfaces in ABS, is that they do not support type parameters. This can be overcome by the use of algebraic data types with type parameters for data structures. For other purposes not easily solvable on the functional level, multiple non-parameterized types is the only available choice.

2.2.2 Classes

Classes are the specific implementations for objects. Thus they typically implement one or more interfaces, although they are not required to implement any interfaces and have publicly available methods. There are no explicit constructors in ABS, instead classes can take parameters which are member variables that need to be specified upon instantiation, similar to how some fields are often specified in constructors. In addition classes can have an *init block* for initialization purposes, which is automatically run upon instantiation after setting

parameter fields and initializing other fields to defaults appropriate for their type.

Additionally one may define a *run* method that is automatically executed after initialization has completed. This is called active behavior, and the run method is added to the set of processes to execute on the object. ABS processes is covered in the next sub-section.

2.2.3 Concurrency

Objects have attached processes, separated into active and passive behavior. Objects with active behavior are those that are of a class that has a run method, which is immediately invoked in a process attached to the object after instantiation finishes. In addition, passive behavior are method invocations coming from other processes. Any method invocation will attach a new process for executing the method to an object unless the method is invoked from a process in the same object.

Every COG can only have a single running process. That is, one object is active, and only one of the processes in the active object is executing; while all other processes in objects within the same COG are waiting to run. A process will yield execution to other processes if it awaits a guard, ABS's conditional waiting, that is if it waits for an asynchronous method call to finish executing or for a certain state to be achieved. This is similar to awaiting a condition object in a loop as long as the desired state is not achieved. As long as there are processes left waiting to run in a COG, they will be scheduled.

When an object is instantiated, it is placed in a new COG or in the same COG as the object whose process instantiates it. Whether the object should be placed in the same COG must be specified explicitly in the source code. While ordinary instantiation places the object in a new group, the programmer can specify that the object is local with `new local Object()`.

In the rest of this thesis, I will refer to ABS processes as tasks, to make a clear distinction between ABS and Erlang processes. There are Erlang processes for every ABS process, but there are many other Erlang processes as well, which we will see in later chapters. The term task is used, because it happens to be the name of ABS processes used in the Erlang back end source code.

2.3 Compiler

The compiler is built into two tiers. A frontend built on JAstAdd, which compiles an ABS specification into an intermediate representation, an object-oriented abstract syntax tree. The frontend does lexical, syntax and semantic analysis. The generated syntax tree is then used by the back end, which will compile the internal representation into a back end language. Code generation is plugged into the modular compiler back

end, making it possible to build multiple different language back ends. The currently supported back end languages are Java, Maude, Scala and Erlang.

2.4 Other tools

In addition to the compiler and back ends for running simulations, there are tools available to do analysis of simulations in some of the available back ends. Sequence diagrams of executions in the Java back end can be produced by connecting a version of the SDEdit tool to a simulation. The Maude back end allows outputting the state of the entire model during simulation.

One of the really useful analysis tools, is the cost analyzer. It makes use of specific annotations, a feature that hasn't been covered, to analyze the expected resource usage of an implementation. The tool is one of the ABS plug-ins for Eclipse.

Chapter 3

The Erlang Back End

ABS has an Erlang back end, which consists of a code generator for the ABS compiler suite and a runtime back end. In this chapter, I will first give a short overview of some of the features of Erlang, and finish with a description of the back end. For a more in-depth coverage of the features, syntax and use of Erlang, readers are referred to Armstrong [2].

3.1 The Erlang Programming Language

Erlang is a functional language made for distributed, fault-tolerant systems. It was designed within Ericsson, and is characterized by the requirements in telecommunications switches; supporting distribution and recovery from faults in one or more processes in a distributed context. This can be used to ensure systems have little if any downtime, which is important in telecommunications and many other server systems.

Because the language is functional, data is in general immutable. Data structures in Erlang are therefore also immutable, or persistent, and changes to a data structure creates a new version of it, which may reuse parts of the old data structure. ABS also uses algebraic, immutable data, making for a close similarity between ABS's functional aspects and Erlang.

Tuples, an algebraic sum type, is used for implementing most of the data structures. Erlang has several other first-class types such as numbers, process IDs, lists and *atoms*, which are a kind of symbolic constant similar to algebraic constructors that take no parameters in ABS.

3.1.1 The Actor Model

The actor model is used for Erlang's concurrency implementation, that is programs consist of processes that communicate by asynchronous messaging. It shares many similarities with the model used in ABS, which is based on active objects. Erlang's processes are lightweight, but

unlike threads, which are often called lightweight processes, they have separate heaps. The processes are lightweight in the sense that they are not implemented as OS threads or processes, but are scheduled by the Erlang virtual machine which does not require context switches.

Message passing is explicitly handled by the programmer, and any value can be passed as a message to other processes. ABS on the other hand has implicit messaging used for calling methods and returning from methods. However, Erlang's standard library contains general purpose modules that handle many of the use cases for messaging, like remote procedure calls. This can relieve the programmer of re-implementing common patterns of messaging, and the use of function calls resembles implicit message passing.

All processes have a mailbox that they may read from using the `receive` expression. When a message is sent to a process, it is added to the mailbox. A process can selectively read specific types of messages, skipping other messages temporarily, just like a person may choose to open personal letters before reading ads or checking bills. If a message does not match the kind the process attempts to read, it is added to a buffer of skipped messages. These are added back to the mailbox after the expression has completed from finding a matching message, or from a timeout specified by the programmer with an `after` block.

Erlang guarantees that messages from one process to another, are received in the order they are sent. However, if multiple processes are involved, it does not guarantee the order of messages between multiple pairs. For instance, if process A and process B sends a third process, C, messages, it guarantees that the order of all messages from A to C is maintained. But messages from both A and B can arrive intermingled. In other words if A and B both send a single message, there is no guarantee that the message from A arrives before the message from B, even if B sends its message after A. The same is true for messages from one sender to multiple recipients: if A sends a message to both B and C, there is no guarantee that B receives its message before C or vice versa.

Fault tolerance is achieved by allowing processes to monitor other processes for failures. Instead of adding error handling in a defensive manner to every function in a program, processes can crash upon errors, and error handling can be delegated to a monitoring process. It is easier to make a single or a few pieces of error handling code, than extensive error handling everywhere. The errors are received by the monitoring process as a message describing the error and which process failed. A common way to handle errors is to restart the failed process or group of processes to ensure continuous operation.

3.1.2 Pattern Matching

Pattern matching[7] is used extensively in Erlang. Function definitions use patterns for parameters, similar to languages like ML and Maude[13]. Variables are bound through pattern matching with the =

operator, which matches the left and right operands, and binds the left operand if it is an unbound variable. Erlang's if-expression is a pattern matching expression, matching one of a selection of Boolean expressions to `true`. The case expression matches one expression to one of a group of patterns, similar to ABS's case expression. Messages are matched in the receive expression in a similar way to the case expression.

3.2 Mapping ABS to Erlang

An executable program compiled from an ABS model, consists of two parts: model-specific code generated by the compiler, and general runtime code. The compiler and general runtime code makes up the Erlang back end, and is the subject of the thesis by Göri [8], which is to be submitted to the Graz University of Technology in 2015. The design described here is based on a draft of his thesis and the source code he has written.

3.2.1 Algebraic Types

ABS's algebraic types are translated into atoms and tuples. If the constructor has no parameters, they are simply represented as atoms with the prefix `data`. When a constructor takes arguments, they become a tuples with a similar constructor atom as the first element, and each argument as subsequent elements in the tuple.

The implementation does not carry with it information about types and their constructors, but relies on the compiler having type checked expressions and statements during its semantic analysis. If values cannot have the wrong type, no type information is needed at runtime, as the code will not bind any values that have the wrong type to arguments or variables.

Some types like Booleans and integers, which are specified as built-in in ABS, are simply represented by Erlang's own versions of those types. Thus the compiler is able to represent any ABS algebraic type using atoms, tuples and types built into Erlang.

3.2.2 Objects

Objects are implemented as state machines. These are a special kind of process with an underlying implementation in the OTP libraries. The library implements the generic part of the FSM, handling message passing, synchronization, and function calls based on messages. The messages passed using the FSM library are called events, as opposed to ordinarily sent messages. First I will give a brief description of the generic part of the object implementation, then I will cover the code that is specific to individual classes and the representation of object references.

The object module that has been implemented, contains functions for each state. These functions are called by the generic FSM to handle the

events that may be passed to the machine. In the case of ABS objects, these are new tasks being created, tasks retrieving or setting field values, committing changes in the object's fields when a task reaches a synchronization point or rolling back changes to the object's fields if an error occurs in a task.

There are also functions for handling events that can occur in any state and usually have common handling across states. Prior to the implementation of the garbage collector, the only event handled this way is terminating the state machine. Messages sent with the standard message passing interface can also be handled by its own function; when a task crashes, objects receive these kinds of messages.

Classes

The general state machine above is identical for all objects, however, ABS has classes of objects, where the classes describe the functionality specific to those objects. The code generator builds an Erlang module for each class, named which contains functions for each method and some standard callbacks that the object state machine uses to handle mutating, accessing and initializing field values. It also generates functions the init block and two general functions for creating field get and set events for the state machine.

Object References

Object references that can be used by the code generated by the compiler, must contain the following:

- A reference to the state machine process, to be able to generate events in the object.
- The name of the class module representing the object's class. Without it the object's methods cannot be called.
- A reference to the COG the object belongs to, to ensure scheduling of tasks in that COG happen sequentially.

All of these are organized into a tuple, to represent object references.

3.2.3 COGs

COGs are implemented as processes that handle scheduling of tasks running in the scope of objects within that group. They can be seen as a sort of state machine, although they are not implemented using Erlang's generic finite state machine module. The state is represented as a tuple, which is passed to a single function, `loop`, that is implemented using pattern-matching. When messages arrive, a new state can be generated based on handling the message, and that state can be entered. A more detailed discussion on the COG state machine can be found in

subsection 5.1.1 of chapter 5, where the necessary changes to support garbage collection, are also covered.

The COG's scheduling procedure, ensures that only a single task is running at any given time. This is done through the use of a token, passed to tasks to allow them to run. When a task reaches a synchronization point, the token is returned to the COG and it allows scheduling of other tasks.

3.2.4 Tasks

ABS processes, or tasks, are naturally represented as Erlang processes. All tasks have a similar life cycle: create an initial state, report to the COG that it is ready, and wait to receive the token from the COG. Upon receiving the token, the task may start execution, returning the token to the COG at synchronization points and upon finishing execution. When the task has finished, it also notifies any processes waiting for it to complete. This feature is currently only used for the main task.

Asynchronous method calls may block while initializing their state, if the object they are to execute a method on, has not been initialized yet. As part of finishing execution they send the result of the method's execution as a message to the future assigned to holding the result.

3.2.5 Futures

Futures are represented as processes that serve the result of tasks after the task has been completed. The future also creates the task on the COG that the callee belongs to, monitoring the COG for crashes until the task has been added to the scheduler. If the COG were to crash before the task could be created, an error will be stored in the future instead of the result of the task's execution.

Until the task has finished, the future will simply wait for the result of the task, not responding to other messages. As soon as the result becomes available, any requests to get the result are handled. If the task crashes, the error is stored and served by the future instead.

3.2.6 Statements and expressions

Most statements and expressions map easily to Erlang, as there are similar Erlang expressions, e.g. `case` and `if`. However, assignments are not so simple, because Erlang only has binding values to variables, that is single assignment. Therefore a unique name has to be used for every assignment. To do this, a counter is added to all variables in the current scope. Erlang does not have a local scope for branching expressions, the scope of variables are function-wide. As a result, the numbers for each variable have to be unified across branches, to the highest number of that variable across the branches. The value available at the end of each branch, is then assigned to the unified variable name at the end of the branch, unless the branch had the

same count as the maximum. Then the high number can be used for generating accesses regardless of which branch was executed.

Loops are also problematic, as there are no loops in Erlang. Instead the functional equivalent of tail recursive functions are used, that is functions that call themselves as their last operation. Loops in ABS, must therefore be translated into a tail recursive function, but an unnamed function does not usually have access to call itself. A small trick is used to ensure this is possible: a function that takes the recursive function as an argument, is used for the initial application of the recursive function with the recursive function as an argument to itself. A sketch of such a set of functions is given in listing 3.1.

Listing 3.1: Sketch of a conditional loop in Erlang

```
Values = ((fun (Inner) ->
             fun (Params) ->
               Inner(Inner, Params)
             end
           end)(fun (Self, Params) ->
                  case loopCondition() of
                    true ->
                      NewParams = doStuff(),
                      Self(Self, NewParams);
                    false ->
                      Params
                  end
                end))(OldValues)
```

Note that the outer-most function in the group, takes only the looping function as an argument. It then returns a function that takes all local variables as a parameter. This function in turn calls the looping function, with the looping function and the variables as arguments. Because the looping function now has a reference to itself, it is able to call itself continuously.

Chapter 4

Garbage Collection

In this chapter, I will give a brief introduction to garbage collection and garbage collection algorithms in general. The material here is mainly based on the book by Jones, Hosking, and Moss [11], which readers are recommended to consult if they need a deeper understanding of the techniques described here.

Garbage collection is the process of automatically freeing memory allocated for data when it is no longer going to be used. It is in general not possible to know which references will be accessed ahead of time. Instead garbage collectors identify accessible and inaccessible data by analyzing what references exist globally or in the scope of one of the program's execution stacks. If there are no references to some piece of data, it is no longer possible for the program to access it in the future, and it can be safely considered garbage and expunged from memory.

Most modern programming languages like Erlang and Java have garbage collectors. This frees the programmer from the burden of explicitly handling allocation and freeing memory as is common in low level languages like C. When memory is allocated for temporary data, but never freed, will result in increased memory usage throughout the lifetime of a program. This is a common bug when programmers are responsible for handling memory, known as memory leaks, and can be hard to track down. The other common problem is known as dangling pointers, where references to data that has been freed and is no longer available, are still in use, which can lead to crashes or, worse still, data corruption if the same memory area is allocated for something else later. Automatic memory management seeks to eliminate these kinds of bugs.

It is common to view objects as a graph, where references are edges and the objects are vertices. Most collection schemes involve a graph traversal algorithm, followed by a run through all objects to free the memory used by inaccessible objects. Collectors that traverse the object graph are known as tracing collectors. The choice of traversal algorithms may affect the order in which memory is read, which affects paging and caching. It is preferable to access memory linearly, or at the very least without moving between pages or cache lines. Objects that are referenced from an object are likely to be allocated around the same

time and be located close together, which means a depth-first traversal is likely to traverse memory in a close to linear fashion. The stage at which the collector actually frees memory, varies greatly between collector types.

A garbage collector typically runs in a separate thread from the program. The program threads are known as mutators in garbage collection terminology, because they alter state. It is generally simpler to do garbage collection by stopping the mutators before the collector goes to work, known as stopping the world. This ensures that the mutator cannot change references while the collector is working, which could lead to problems where the collector loses sight of accessible objects due to changes in the object graph.

4.1 Reference Counting

Some non-tracing collectors, which can be simple to implement, use a technique known as reference counting. Reference counting means that whenever a variable is set to reference an object, that object's count is incremented. Likewise it is decremented when the variable is changed or moves out of scope. If there are no more references to an object, it is considered garbage and can be collected because there is no way to access it.

Because the mutator is actively incrementing and decrementing the counts, it may also be responsible for freeing the occupied memory. Although, it is common to run a separate collector thread, particularly as part of taking measures to handle cyclic garbage. Leaving the responsibility of collection up to the mutators, can be considered a simple form of concurrent collection, i.e. collection that runs simultaneously with mutators, but it can be argued that the mutator is not performing work while it is freeing memory.

Yet, collecting garbage in multithreaded and particularly in distributed programs can be problematic. One must protect against increments arriving late, i.e. after decrementing the count to zero, which could lead to premature collection of objects. First atomic increment and decrement operations are necessary when multiple threads may alter the count. Then if a reference to an object is passed to a different node, the passing thread must increase the count, to ensure that the passing thread does not decrease the count to zero before the receiver can increment the count.

The Java EE distributed collector uses a variation of reference counting known as reference listing, where it considers sets of nodes that may have references to objects. Long delays are added before collecting objects, due to the possibility of latency in the communication with nodes that have references, obscuring these nodes. This leads to garbage staying uncollected for a long time.

Reference counting cannot handle object graphs containing cycles without adding some kind of special handling of this. One option

is to have the programmer account for it using what is known as weak references, that is references that are not considered by the garbage collector. There are also algorithms that attempt to determine automatically if a reference should be considered weak or strong, but they may incorrectly deem some references weak, leading to premature collection of some objects.

Another option is to use a hybrid collector, using a tracing algorithm to take care of cyclic garbage. The tracing collector can be a full tracing collector that is only run rarely. This yields fast operations most of the time, but longer delays when the tracing collector runs, which can be undesirable if predictable pauses are required.

A full trace may not be necessary, as the cycles in the object graph usually make up a small sub-graph. Therefore it can be sufficient to only trace the sub-graph, starting in a node that has had its count decremented. If the node can be traced back to itself, and no other references to it exists, a garbage cycle has been found.

4.2 Mark and Sweep

The simplest form of collector that actually traverses the object graph to account for whether objects are accessible, is the mark and sweep collector. It consists of two phases, one for traversing the graph and marking the objects that are encountered, and one for freeing, or sweeping, the garbage.

The marking phase uses a tri-coloring system similar to other graph traversal algorithms that avoid processing vertices multiple times using a two or three colors. The colors or marks are used to denote three sets of objects. Objects colored white are objects that have not yet been encountered, all objects start out white. Gray objects have been encountered before, but have not been processed for references. Those colored black are objects that have been processed, meaning all objects they reference have been colored gray.

The marking phase starts by processing the root set, which are usually made up of the references from mutators' execution stacks, but programming languages and software systems may define additional roots. Objects currently in use by one or more mutators cannot be collected, and makes for a root, or starting point, of the graph traversal. Finding the roots requires processing the stack and identifying the references. Similarly when processing references in an object, the references it contains must be identified. The identification process is largely language dependent, as different languages represents references differently.

After the roots have been marked gray, or added to the gray set, they can be processed for references. The object is colored black, then all references in the objects are followed, and all white objects referenced are colored gray. When the gray set is empty, there are no more objects to process, and all white objects can be collected. Pseudo-code for

the algorithm is given in listing 4.1. Note that it is common to use variables to denote sets in the objects instead of applying set operations like union, intersections and complements. In this example Δ is the universal set of the garbage collection domain, i.e. all objects.

Listing 4.1: Pseudo-code of the mark and sweep algorithm.

```
Gray := findRoots()
While Gray ≠ ∅ Do
  X := getElement(Gray)
  Black := {X} ∪ Black
  Gray := (Gray ∪ references(X)) \ Black
End

White := Δ \ Black
sweep(White)
```

Sweeping consists of running iterating over the heap, and freeing all white objects. This leaves the memory area fragmented, as holes are scattered throughout the heap. When allocating memory for new data, these spaces may be hard to utilize, and different allocation strategies have to be considered. Figure 4.1 illustrates the results of collecting garbage by sweeping.

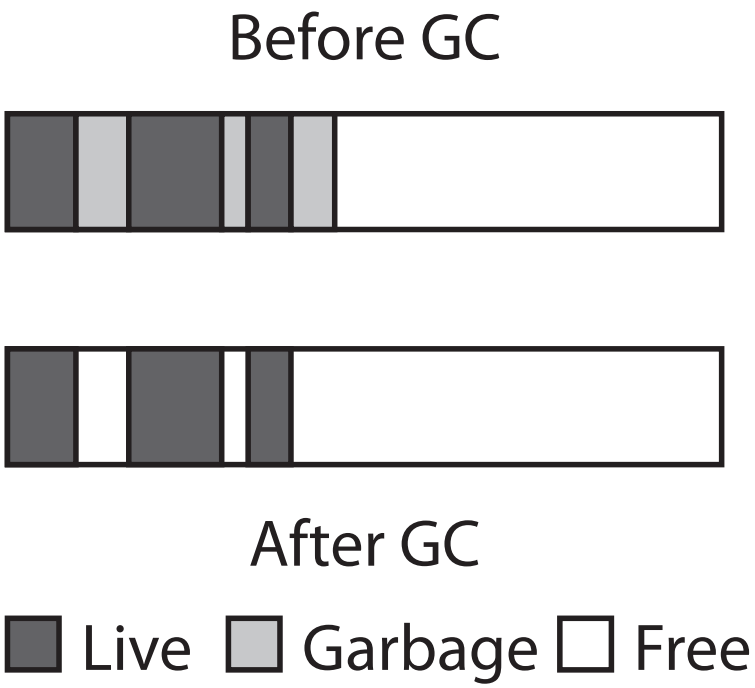

Figure 4.1: Heap before and after performing a mark and sweep collection.

4.3 Mark and Compact

The mark and sweep algorithm can leave memory fragmented as free memory is not necessarily a contiguous area. This can lead to slower allocation due to searching for blocks of suitable sizes for new data. It may even require increasing the heap size, even though the sum of free memory on the heap is large enough to fit the new data. Instead of simply freeing the objects that are garbage, leaving holes of different sizes in memory, it is of interest to ensure that free areas are kept contiguous. Compacting is the process of moving the accessible objects to one side of the heap, overwriting the objects that are no longer needed.

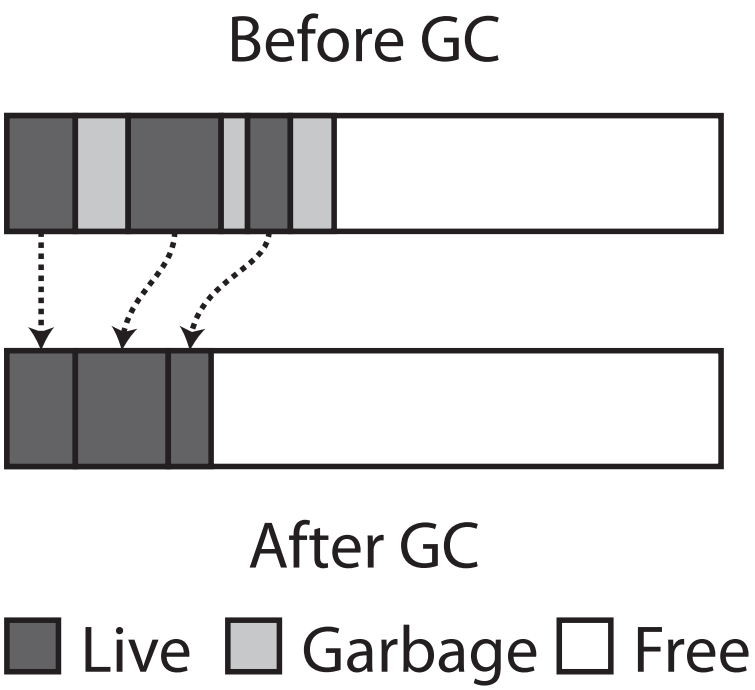

Figure 4.2: Heap before and after performing a compacting collection.

Because objects are moved, there's a need to change references to reflect the new address of the object. This requires that the new address is stored temporarily, and that all references are updated. It is therefore common for compacting collectors to iterate over the heap multiple times, although it is possible to do it only once by keeping forwarding addresses in a table.

4.4 Copying Collectors

Compacting can be quite slow, compared to sweeping, requiring multiple traversals of the heap. It does have the desired property of fast allocations, which the fragmented heap left over by a sweeping collector, hinders. Copying collectors achieves similar compaction, but splits the heap in two semi-spaces. When collecting garbage, all non-garbage is copied to the other half of the heap. There's no danger of overwriting objects in the *from space*, the part of the heap the collector is copying from, in the process of copying objects. The forwarding address can therefore be stored safely in the space the object used to occupy, allowing updating references without the need for additional traversals. This yields faster compaction at the expense of requiring twice the heap space of a normal compacting collector.

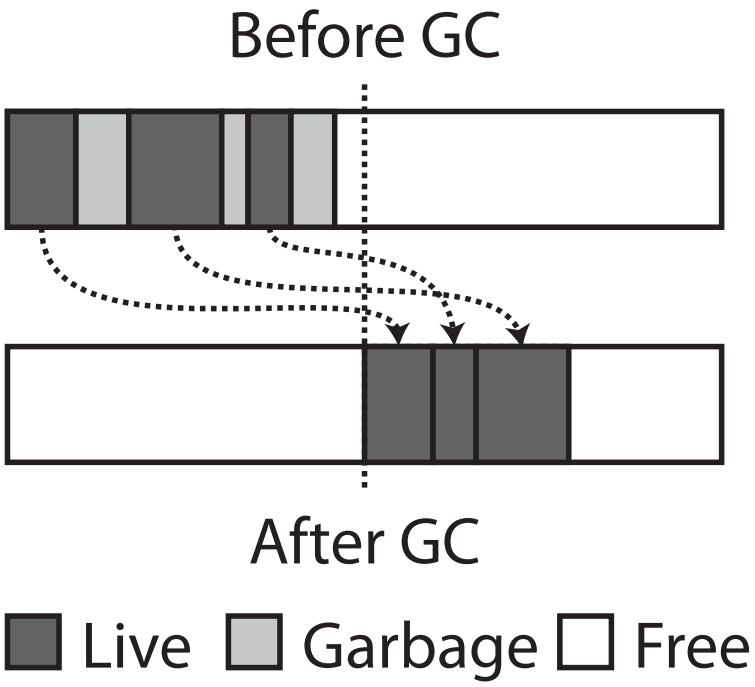

Figure 4.3: Heap with semi-spaces before and after copying collection.

As is illustrated in the figure above, a copying collector is free to reorder objects in memory. Thus a copying collector can optimize the memory layout of objects to help mutators with caching and paging.

4.5 Generational Collectors

A copying collector has to copy all black objects every time a collection cycle completes (sometimes there are objects that cannot be moved, but

I will not cover this here). This can lead to copying taking unnecessarily long. Marking a large amount of objects can also take long. Instead, one can partition the heap based on the longevity of objects, avoiding having to mark and copy objects that are unlikely to be garbage. Long-lived objects can be moved to a partition that holds only long-lived objects based on surviving garbage collection cycles. A partition for old objects, doesn't have to be considered by the garbage collector at every collection cycle.

For different *generations*, different garbage collectors can be used. If a slower collector is used for an old generation, it may not affect runtimes much, as this generation is not collected as often. There is one issue with only looking at a smaller part of the heap at a time, and that is references between objects in different generations. Some additional overhead is added, because there's need to track these. But this overhead should be low compared to the benefits of a partitioned heap.

4.6 Distributed and Concurrent Garbage Collection

Concurrent garbage collection is when mutators are allowed to run alongside the garbage collector. It has also been called parallel collection, but that term is usually used for multithreaded garbage collectors. Because concurrent collectors do not stop the world for the duration of the collection cycle, mutators may change references while the garbage collector is tracing objects. This can lead to what is known as the lost object problem discussed in subsection 4.6.1.

Distributed collectors are collectors used in distributed systems. Often one collector runs on every node in the system, requiring collectors to come to a consensus. One example of a distributed collector is the mark and sweep collector in Emerald, which uses a 2-phase commit to ensure consensus [12]. However, distributed collection can also be done by a single global collector, that handles garbage on all nodes. This is slower and can require more memory for the collector to have a complete view of objects across nodes.

Distributed collectors are sometimes designed specifically for certain models of distribution, like active objects or actors. One such collector found in [3], uses frequent, evenly spaced messaging between active objects to detect garbage. It has an interesting approach to cyclic garbage, where the active objects forming the cycle, come to a consensus between themselves to determine that they are garbage.

4.6.1 The Lost Object Problem

The invariant used in tri-coloring, is that no black objects have references to white objects. This holds because an object that is black, has had its references added to the gray set. Marking is monotonic in

the sense that objects that have been colored gray, can never become white, and objects colored black can never become gray or white. This is needed for the invariant to hold, and can be stated simply as “no object object can become lighter” [5]. The monotonicity property is necessary to ensure the invariant holds, as if a gray or black object could become white, obviously a non-white object pointed to be a black object could become white.

If mutators are running simultaneously with the garbage collector, they may alter the object graph by assigning references to fields in objects. Should a black object have one of its fields set to reference a white object, then the invariant above ceases to hold. This is a problem, because other references to that white object may be deleted, although there is now a reference to the white object in an object that will not have its references traversed by the collector in the current marking cycle. Figure 4.4 illustrates how an object, C, could become invisible to the garbage collector, if a mutator copies the reference and deletes the reference it copied it from, with code such as : `A.C = A.B.C; A.B.C = null;`

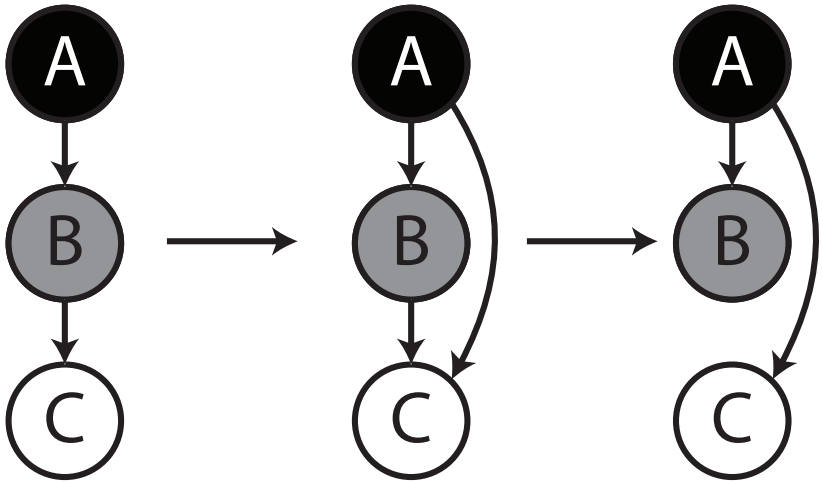

Figure 4.4: The lost object problem.

A concurrent collector has to protect against this. One way to do this is to apply a write lock to black objects. Disallowing changes to its variables if the newly assigned value is a reference to a white object. The referenced object, would first have to be added to the gray set. Another option is to make gray objects read protected with a lock. No variables in a gray object could be accessed without first adding them to the gray set. This can be done as a conditional wait, waiting for the object to become black before accessing it. Marking of the gray object the mutator is trying to access, can be prioritized by the garbage collector in its marking strategy as in [12].

Chapter 5

Implementing Garbage Collection for the Erlang Back End

The Erlang virtual machine, BEAM, has a concurrent copying garbage collector [16], but ABS's Erlang back end uses long-lived processes for every COG, object and future. These processes will not be stopped until the entire simulation is complete, that is if there are no more tasks to be scheduled on any COG, or optionally if such processes encounter errors and crash. Since these processes are not stopped, any data they reference cannot be collected by Erlang's garbage collector, as it is still available to a process.

There is also a limit to the number of processes that may exist in the VM at any given time, although this is configurable and can be set higher or lower than the default of $262,144$ (large parts of the documentation still indicate the default is $32,768$, but the number can be found in the emulator manual). Even if memory consumption is not an issue during a simulation, the system may create too many processes leading to a crash.

The garbage collector's task will therefore be to stop the processes that represent COGs, objects and futures, which will allow Erlang's collector to free the memory used by them. Stopping these processes, allows new objects to be created; whether it releases the memory required for it, or just lowers the number of running processes to keep within the VM's limit.

The garbage collector that has been implemented in this thesis, is a mark and sweep collector that stops the world as described in section 4.2. Algorithms that rely on moving objects, are not applicable, since we're collecting processes and not ordinary objects. Processes do not have memory locations that we may access or move. Erlang's own collector will handle the collection of the memory occupied by the process.

The processes that implement object state have references to the COG responsible for scheduling tasks on that object. Because COGs

are not first-class citizens in ABS, they can only be referenced by the objects they contain. Therefore, simple reference counting can be used for COGs.

Mark and sweep could be used for COGs as well, but has been restricted to objects and futures. Using the mark and sweep collector would add more elements in the messages sent to the garbage collector when it retrieves references, and more memory used by the garbage collector as it builds sets of gray and black objects. Using reference counting for COGs requires only minimal changes to handle messages regarding the creation and termination of objects.

It should be possible to implement a concurrent garbage collector for the back end, and this will especially be of interest if the back end is made distributed. Stopping the world was chosen because of the limited time frame of this thesis. This avoids some of the added complexities of adding locks or other protections against the lost object problem.

5.1 Stopping the World

Stopping the world involves pausing every mutator while the garbage collector performs a collection cycle. After the cycle is complete mutators can be resumed. Because the garbage collector is implemented on top of a virtual machine without access to low-level instructions like interrupts, mutators cannot be stopped arbitrarily. First we shall see how the back end can be changed to stop COGs from scheduling tasks, and then I will present the problems that were found and solved to ensure that the garbage collector stops mutators correctly and does not collect anything that is not garbage.

5.1.1 COG States

Before the addition of the garbage collector, COGs could be in one of the following states: *no runnable tasks*, *not running* or *running*. If there are no runnable tasks, the COG waits for new tasks or a change in a waiting task. Regardless of what kind of change occurs, it transitions to the *not running* state, to attempt scheduling a task. In the *not running* state, it will still check for incoming new tasks or changes to tasks' states, but will immediately try to schedule a task if there are no such messages waiting in its inbox. When there are no tasks to schedule, it will go from the *not running* state to the *no runnable tasks* state. If there is a task running on one of the COG's objects, it will continue running until that task reaches a synchronization point or fails. The COG will still be able to add new tasks to its set of tasks or update the state of waiting tasks. The described machine is illustrated in Figure 5.1.

There is a need for an own state where the garbage collector is running, that does not allow scheduling tasks, i.e. a stopped world state. To do this, we add a state where a record is kept of the state the COG should enter when it is resumed after garbage collection is finished. The

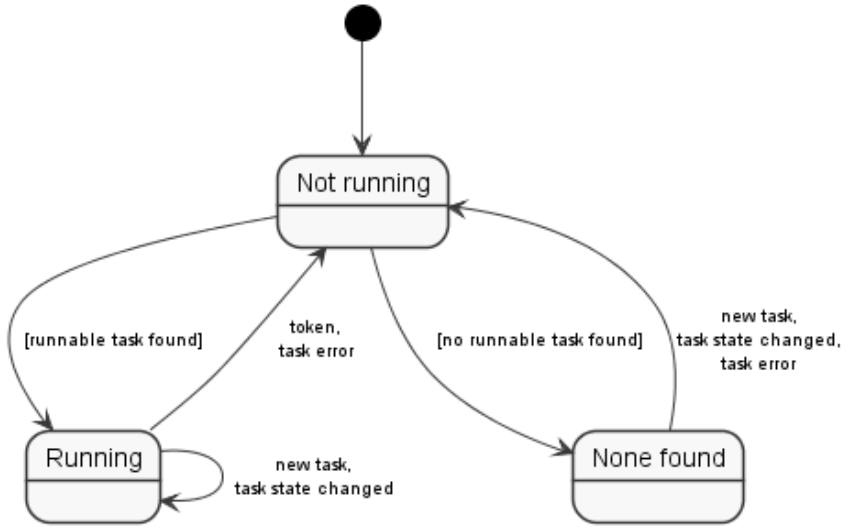

Figure 5.1: The COG finite state machine before the garbage collector was implemented.

stopped state will be entered if a message from the garbage collector to stop the world is received. However, the only obvious places where the world could be stopped, is when no task is running, i.e. when tasks have reached synchronization points and yielded execution, which may include having finished running.

The rest of sub-sections will discuss issues with stopping the world, and stopping the world at other points than the standard synchronization points. We will see yet another state added to the COG state machine, leading to a machine as shown in Figure 5.2.

5.1.2 Tasks Blocking on Futures

Tasks that are trying to get the result of a future, do not yield execution, but will block until the future is resolved. This leads to possible deadlock situations in a few scenarios in ABS. Two cases of logical errors in models can result in deadlocks. If a task blocks on a future associated with a task on a local object, that future can never be resolved, because only one task can run in a COG at a time. If a task blocks waiting for a task on another COG, and a task on the other COG is blocked, a deadlock can occur. If they are mutually blocking on futures with tasks on each other's COGs, it is an error in the model. The task could also be waiting for a task to finish on a COG that has been stopped by the collector. The latter situation would lead to the garbage collector waiting for the COG to stop, while that COG cannot stop before the future is resolved. The future cannot resume until garbage collection is complete. Then the world cannot stop.

To handle this situation, we add an additional state to COGs for blocked tasks. If a task is blocked, no tasks can be scheduled. When the

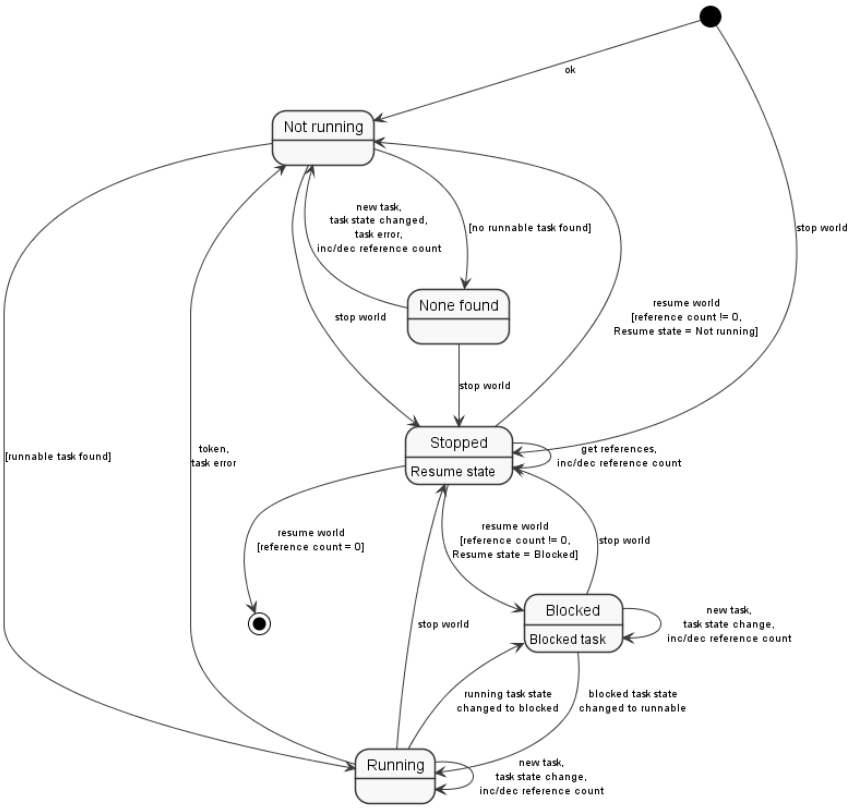

Figure 5.2: The COG finite state machine after the garbage collector was implemented.

future the task is waiting for, is resolved, the COG can schedule that task again and return to the running state. While a task is blocked, the COG can be stopped and the task will respond to messages from the garbage collector to get its local variables. The additional state is shown in Figure 5.2.

5.1.3 Stopping Running Tasks

Long-running tasks pose a potential problem if they do not contain synchronization points. Recursive functions or methods, other long chains of synchronous method calls and loops are all problematic in that they slow down stopping the world considerably. The worst case would be an infinite loop such as: `while (True) { new Object(); }` There are no synchronization points or any blocking statements as part of this loop, which would lead to the collector waiting indefinitely for this mutator to finish. To deal with this situation, there is a need to be able to stop running tasks, without yielding execution in favor of other tasks and without slowing down execution time considerably.

By checking the task's mailbox with an instant timeout for a stop message, the delay imposed by the check will be kept minimal. Task's are only sent messages when they are in non-running states, except for

the stop messages introduced here and a notification message for the main task that is kept in the mailbox until it completes execution. This ensures checking the mailbox runs in constant time as there is at most one message to skip before reaching the end or the stop message. This is subject to change if future revisions of the Erlang back end add more observers waiting for notifications of tasks completing.

It is desirable to respond quickly to stop messages, but to add as few such checks as possible. Checking for stop messages between every expression would be easy to implement, but would add excessive delays. The other easily implementable option, would be to add these checks to the beginning or end of functions, methods and loops. Adding them to the beginning seems more appropriate as the ends are often followed by tasks ending or reaching synchronization points in an outer scope. By placing the check at the beginning, any function application, method call or loop iteration can be immediately stopped.

5.1.4 Asynchronous Method Calls on Inactive Objects

When a method is called asynchronously a new task is created. The task will not finish its own initialization until it receives a message from the object that the object has been activated. An object is active once its init block has completed execution, and its active and passive behaviors can be allowed to start running. However, if the object's COG has been stopped, initialization cannot complete, and the tasks will block until the world is resumed.

Although, the COG has been stopped, the tasks for executing the asynchronous method calls have not been initialized yet. The task registers with the object using the finite state machine API's synchronous event. Because this is a synchronous event, the sender has to wait until a result is produced. An FSM is not required to complete event handling at the same time as it starts handling the event. Instead the object FSM keeps track of all tasks that are waiting for it to become active, and produces an event result for all of them when it does.

This blocking is problematic because tasks make up the mutators, and as such their execution stacks contain parts of the root set. The garbage collector will therefore try to retrieve references from these blocked tasks, but they are unable to respond to any messages until they receive responses to the events they produced for the object FSM.

A change in the blocking behavior is required, so they also accept messages from the garbage collector. There are two options for how this can be achieved. One is producing a result for the event whether the object has become active, and if it is not yet active, wait for a message from the object or the garbage collector. The other is to use an asynchronous event and wait for messages just like in the case of inactive objects in the former technique. The downside of asynchronous events, is that they do not produce any errors if the FSM has stopped, like the synchronous events do. Therefore I have chosen to keep the synchronous event, and instead handle the event result according to

the state reported.

5.1.5 Tasks Created While the World is Stopping

If the world is in the process of being stopped, some tasks may continue to run for a while until they reach a synchronization point. While they are running, they may produce new objects, including new COGs if the objects are not local, and new tasks. If a reference that is only available as a local variable in a task is passed on to a new task and then the first task returns, the reference could be lost due to a race condition. A short ABS example is shown in listing 5.1. If this method finishes execution before the task to perform `anotherMethod` has been added to `someObject`, there will be no references to `anotherObject` in the part of the graph visible to the collector.

Listing 5.1: A method that illustrates the lost task problem.

```
Unit someMethod(SomeType someObject) {
  SomeOtherType anotherObject = new local SomeOtherClass();
  someObject ! anotherMethod(anotherObject);
}
```

When an asynchronous method call is made, first a future process is created. The calling task receives a PID for the future, which it passes on to the garbage collector to inform it of the future's existence. The future then sends a message to the COG of the object the method will run on, to create the task. If scheduling and/or latency delays this message, all COGs may have been able to stop in the meantime. The garbage collector could then retrieve references before the task is created. The retrieved references would not include the parameters of the yet to be created task, leading to potentially sweeping objects that will later be used in the new task.

Before the task itself is created, the future for holding the result is created in the calling task. This future will be registered with the collector when it is created, which happens before the task finishes or reaches a synchronization point. The calling task's COG cannot stop until the task reaches such a point. If the future is considered a root by the garbage collector until the task has been created successfully, the parameters passed to the task can be made available from the future.

Message ordering, however, is not guaranteed. The calling task first sends a message to the garbage collector informing it of the future's creation, before it may communicate with the COG to allow it to stop. The COG cannot possibly send its message to the collector before the task does, due to the COG waiting for a subsequent message from the task before it can stop. This gives a kind of weak guarantee that the two messages will be received in order, as long as the process of multiplexing messages into a stream on the sender side, and demultiplexing the stream into messages on the receiving side does not re-order messages. Although it is unlikely the order of these messages can be switched, Erlang does not provide guarantees that they will not be re-ordered.

It is desirable to have strong guarantees, however, which means future creation must be synchronized with the garbage collector. This requires waiting for a reply from the garbage collector before the task can continue execution after the future has been created.

5.1.6 An Incomplete View of the World

Because the garbage collector depends on messages to register COGs, futures and objects, its world view might not be up to date at all times. There may be COGs that have been created, but are not known by the collector yet. Having a complete view of COGs is important when stopping the world, as without it, some COGs may not be stopped when the collector starts its marking phase. Then parts of the root set will not be visible, and some objects may not be marked as they should. It is equally important that the collector has a complete view of root futures, for the very same reason.

If the collector cannot find all roots, objects could have had every reference to them removed from objects in known COGs and futures. Yet their references may have been passed on to objects on new COGs. With the references out of view from the collector, they could be collected prematurely.

Note that a complete view of objects is not necessary, as they cannot be roots. Should there be objects that remain unknown to the garbage collector in the current cycle, their collection would simply be delayed until a later cycle when they are known.

We can use similar arguments about message ordering as was used in the discussion on lost tasks above. The same reasoning applies to creation of COGs and stopping them. If the garbage collector attempts to stop a COG that has a running task creating new *far* objects, the messages reporting the creation of the COGs for the new objects, will be sent by the task before it may reach a synchronization point. This requires that creation of COGs also synchronize with the garbage collector to guarantee that the world view is complete. The synchronization on future creation above guarantees a complete view of root futures, and with similar synchronization for COGs, we can guarantee a complete view of COGs as well.

While the collector is stopping the world, it must tell new COGs to stop as well. This could potentially lead to an increased wait time before the marking phase can begin, particularly if these COGs are able to start tasks before the garbage collector can send them messages to stop. If the tasks they run create new far objects, which in turn manage to run some code before their COGs are stopped and so on, potentially the collector and all COGs that are stopped could end up waiting indefinitely for the rest of the ever-expanding world to stop. This problem is unlikely to occur in a non-distributed version of the back end. Introducing latency increases the likelihood as the collector could have a slow connection to the nodes the COGs exist on.

One solution to this problem, is to have COGs wait for confirmation

from the garbage collector before they are allowed to start scheduling. COG initialization would slow down slightly from this solution, but it would guarantee that the stop message arrives before any tasks can be scheduled. Note that this is messaging between the garbage collector and the new process representing the COG, and not the same as the task creating the COG synchronizing with the garbage collector. Both messages from the garbage collector can be sent as part of handling the creation event, but to two different receivers: the process representing the COG and the process representing the creating task.

5.1.7 Blocking Object Instantiation

When objects are instantiated on a new COG, they add a task for running the object's *init block*. The instantiation does not complete, until the COG has accepted the task into its queue of tasks to schedule. This is common for all tasks being added to a COGs scheduler, whether they are asynchronous tasks added by a newly created future process, the main task added by the runtime environment or an initialization task added by a task running on another COG.

Both futures and tasks adding new tasks to another COG need to be able to respond to the garbage collector while waiting for the task to be added for scheduling. Unlike a future, a task creating an object, has to make sure its own COG is possible to stop while the task is being added. For this, the blocked state added to COGs is used. The task enters the blocked state, then waits for the new COG to accept the task, before becoming ready for execution again. The new COG may have been entered into the stopped state instantly, as per the communication described in the previous section.

5.2 Marking Objects and Futures

When the world has stopped, the garbage collector can move on to the marking phase, where it will traverse the object graph to separate all reachable objects from unreachable objects. In the implementation, actual sets are used instead of a table of marks or marks in the objects.

The mark phase begins by getting references from tasks and root futures, which represent tasks that have not yet been created/executed. Tasks and root futures are special in that they cannot be garbage collected and as such do not need to be placed in the black set. Instead the first iteration of the mark phase starts with an empty black set, and a gray set consisting of all references from the tasks and root futures. COGs contain lists of tasks, so the work of collecting references from tasks are delegated to the COGs, while the root futures' references are collected directly by the garbage collector. Note that the use of an empty black set is a flaw. The black set should start as the set of all root futures to avoid retrieving references from them twice, should there be references to them in the graph.

In each iteration of the garbage collector, if the gray set is not empty, it will be processed by polling objects in the gray set for references. These references must be cross-checked against the black set, so we do not reprocess objects. When references have been retrieved from an object, it is added to the black set and removed from the gray set. Then the newly discovered references are added to the gray set. The mark phase is over when there are no more gray objects.

Erlang can process a list of objects in parallel, evaluating a function on all the elements and returning a list of the results. Because retrieving references from an object is not dependent on other objects, this step can be done in parallel. If the gray set is large, this should speed up the marking phase. Then the entire previous gray set can be added to the black set, the retrieved sets of objects can be combined into a single set and filtered for elements of the new black set. This way the entire wave front is handled at the same time.

The traversal would then be a breadth-first traversal of the graph, which is often undesirable in low-level garbage collectors because these objects tend to be spaced farther apart in memory. The order should not be as important in the context of the ABS back end, because references are not memory addresses. It may nonetheless affect paging, because it affects scheduling of these processes. The Erlang scheduler will be very active, to handle messages being retrieved and sent by many different processes. This could result in a lot of paging regardless of traversal order, because the garbage collector processes as well as the object processes would be needed in interleaved patterns.

Instead of using sets a mark table could be used and might lower the memory usage of the garbage collector during the mark phase. Changing to a mark table, should be easy, as the set of all known objects could be changed to a key-value store with objects as keys and marks as values. There are some issues to consider if a mark table is to be used, for instance balanced trees have slow and fast functions for most operations depending on whether you know if a key is in the structure already. Which functions to use would depend on whether you could guarantee that all objects are known to the garbage collector before the mark phase.

Yet another option for marking, is to delegate the task to the objects themselves. The garbage collector could initiate marking with a message to all mutators and root futures. They could then pass the message on to all their references. There would need to be some kind of barrier within all objects as they pass on the marks, so they only return after all their referenced objects have been successfully marked. When all mutators and roots have completed marking, the garbage collector could send a message to sweep all white objects and resume the rest. This would require as many messages as there are references in the graph, which would increase the number of messages, but all messages could be kept small. Without experiments, it's difficult to say if this could improve the speed of marking.

5.3 Sweeping White References

When the gray set becomes empty, any object that has not been seen by the collector during the mark phase, may be collected. Sweeping involves sending messages to all the objects that remain white. Because we have a set of all known objects, including non-root futures, and a set of black objects, the white set is simply the black set subtracted from the set of objects: $White = \Delta \setminus Black$. Erlang's set data structures have standard set operations like difference, union and intersection, which are used.

Once the white set has been obtained, all white objects are to be removed. Removal is done by sending a message asking the Erlang process that represents the object, to stop. There is no need for the garbage collector to wait for a reply as we have no further interest in these objects. As there should be no other messages sent from other processes to these white objects, otherwise they would be in the black set, the process should stop reasonably quickly.

Before resuming the world, the new set of all objects must be built. Because the black set may also contain root futures, which should not be collected in the next pass if they remain roots, it cannot be used as the new set of all objects the garbage collector knows of. One of three options exist for the calculation of the new set of objects:

1. Remove root futures from the black set.
2. Remove white objects from the old set of all objects.
3. Intersect the old set of all objects with the black set, which is a variation of the first option.

The effect of the different variations have not been tested. The best option, would be to choose the option that would give the fastest operation. The speed of the operation, depends on the sizes of the sets, although how much it varies depends on the choice of set implementation discussed below. In the implementation, option three was chosen arbitrarily. Because sweeping has been one of the faster parts of the collector, the other options have not been tested.

To perform set operations, sets must be of the same type, either ordered lists, `ordsets`, or balanced trees, `gb_trees`. If the two are of different types, one has to be converted to the other, which adds an additional pass over the data structure, typically with linear complexity. Balanced trees use algorithms of the following complexities $\mathcal{O}(|S|)$ and $\mathcal{O}(|T| \times \log|S|)$ for all set operations, where S is the largest of the two sets T and S. The algorithm chosen, is the one that achieves the faster execution of the two for the given trees.

Trees are slower than ordered lists when the sizes of the sets are almost equal, but have faster insertion times for single elements. This makes them a natural choice for the set of all objects, which have single objects inserted often. If the sizes of the sets are very different, the

set operations are considerably faster with trees. That means, if there are many objects to collect, trees will be faster. However, if almost all objects are black, it will be slower.

Lists have a smaller memory footprint, and make for shorter messages when sending references. It's also necessary to use lists for the parallel mark, without writing the parallel mapping function from scratch. In the end, a combination of ordered lists and trees were used, but this can easily be changed by exchanging the module names, as only functions common to all set modules are used.

5.4 Reference Counting COGs and Resuming the World

The only objects that may contain references to COGs, are the objects that belong to the COG. Therefore a COG can simply contain a count of the number of objects in the group, i.e. it can be reference counted. However, COGs are not aware of the objects they contain, but the objects keep reference of which COG they belong to. When an object is created, a message is sent to its COG to tell it to increase the count. The first object created on a COG, the only object that is not created locally, does not send such a message. Instead the COG starts its count at 1. This ensures that the garbage collector cannot start and kill the COG before its count is first initialized. When an object is killed, either from the garbage collector sweeping it away or from an error in a task, it sends a message to its COG to have it decrease its count.

The messages to increase the count are sent by the task in which objects are instantiated. These tasks run on the same COG as the object being created. Because only local object creation involves an increase message, the COG cannot be stopped before the task sends another message. Message ordering between pairs of tasks therefore guarantees that the reference count is up to date before any garbage collection occurs.

When the garbage collector finishes sweeping objects and futures, it will ask COGs to resume scheduling. The COG at this point should check its number of referencers, and if the number has reached zero there are no more objects in the group, which can therefore stop running. Before it stops it must inform the garbage collector that it has been removed by sending a message. One possible issue here, is if the message does not reach the garbage collector prior to the next garbage collection cycle, the collector may attempt to stop the COG that no longer exists. If resuming the world is synchronous, i.e. the garbage collector waits for all COGs to resume or die before it continues its normal execution, its world view will not include dead COG processes.

Instead of synchronizing resumption, we rely on the message from the COG regarding its death, to arrive while the garbage collector waits for it to stop. Sending a message to a dead process simply leads to the message disappearing, and the collector will wait for a message from the

COG before starting its next marking phase. Therefore we can be sure the COG will be removed from the set of COGs before marking begins in the next cycle, and that marking will be able to begin, as the number of stopped COGs will match COG total.

5.5 Static Analysis

Some changes had to be made to the collector. Particularly generating the representation of the stack to give the collector access to all references in local variables in any of the calls on the stack. Instead of placing every local variable in this representation, it is preferable to try and minimize it by only including variables that may contain references. Therefore variables' types have to be checked for whether they may contain references. Reference types, that is references to objects, necessarily contain references unless they are set to null. Because futures is one of the referencable types the collector collects, any future variable is considered a reference types. Additionally, algebraic types may take reference types as type parameters, or have constructors that take reference types. Therefore all type parameters and constructor arguments are checked for reference types as well.

It is also of interest to avoid blocking unnecessarily. Analysing whether getting future results would block or not, was also added. This can be seen as a variation of an analysis known as the *reaching definitions*, which analyses which assignments of variables, may reach a statement [14]. This variation becomes which assignments of future variables, that have not been awaited, may reach this statement. Instead of using the typical framework for doing static analysis, that is building a set based on a work list, a simple analysis of variable counters was used. Recall the counting used for variables in the Erlang back end which was covered in `Statements and Expressions`.

It could be beneficial to add more static analyses to the compiler, like checking whether functions are recursive or at least deep, to remove being able to stop a task while executing code that will not take long. Similarly loops could be checked for whether they contain synchronization points to avoid bloating them with multiple places where they may be stopped.

5.6 Triggering Garbage Collection

One of the issues of running garbage collection, which will slow down the system somewhat, is when to perform garbage collection. In the case of collecting data, one obvious choice is to collect when there's not enough room for new objects. The equivalent in our case, would be if process limit is reached. The garbage collector doesn't have such fine-grained control, as processes aren't spawned by the collector. The ratio of processes to the process limit, which I will denote Φ, is used instead. If this ratio exceeds a threshold, garbage collection is triggered. The

threshold adapts linearly after every garbage collection cycle, from 50% to 90%. This trigger is more like an emergency trigger, that is unlikely to be used in most simulations, but could become necessary for the most aggressive models.

In chapter 6 different triggers will be tested. They may only be triggered upon events that the garbage collector sees, like futures being resolved, objects or futures being created, etc. Another place to look, when basing collection on the amounts of processes running that it may collect, is threshold's on the number of objects and/or futures. One such trigger was tested, where the threshold doubled after every collection if the count was within 75% of it. The threshold would also halve if the number of objects and futures fell below 10% of it.

A timed trigger was also tested, where the garbage collector would run if an event occurred and 100 ms had passed since the last collection finished. The chosen timeout of 100 ms may very well be too low or too high. Possibly the timer should also be made adaptive, so it could become longer if there's little garbage to collect.

All triggers can be combined, and the baseline trigger based on the process ratio is always there, except when testing never collecting or always collecting. Combining timers and counters, could be a good idea, if the count threshold gets too large to ever collect any garbage, the timer can deal with the situation. So this combination was also tested.

Chapter 6

Evaluation

In this chapter, I will evaluate the implementation according to the goals that were set for the garbage collector:

- Fast - The garbage collector should not invoke considerable slowdowns to the system.
- Comprehensive - The garbage collector should collect all or, at the very least, most identifiable garbage.
- Correct - The garbage collector should not collect objects that may be needed in the future.

Speed is relatively easy to test for, measuring the running time of simulations. Comprehensiveness is somewhat more difficult to show, but may be seen in some test cases from the memory usage and number of objects collected. However, the choice of a mark and sweep collector, ensures completeness as long as there are no unnecessary roots. The choice of roots has been covered in the implementation chapter, and has been kept minimal while maintaining correctness. If the garbage collector collects objects that are needed in the future, crashes should occur occasionally from the use of dangling pointers. No such crashes have been observed in the final version of the collector.

6.1 Metrics

Before presenting the results, I will introduce the different metrics used, how they are obtained and why they are used.

6.1.1 Execution Time

One of the goals set for the collector, is to avoid excessively slowing down the execution speed of simulations. Therefore it is essential to measure execution time. The total time spent running a simulation before and after adding the garbage collector, gives a good indication of the total processing overhead. Some of this overhead will come from

added synchronization, additional use of resources in mutators to be able to respond to the collector's requests, as well as the pauses invoked by the collector during collection cycles.

There are two sides to the execution overhead, the work done by the collector while pausing the mutators and added work for mutators. It is of interest to not only measure total execution time, but also to measure the time spent in garbage collection cycles to get a better view of the overhead. The garbage collection cycle consists of the following phases: stopping the world, marking objects, sweeping objects and resuming the world. The start of each phase of the garbage collection cycle is registered with timestamps of microsecond accuracy. The timestamps marking the start of one phase, marks the end of the previous phase, and allows us to find the time spent in each phase. These intervals give an indication of where the garbage collector spends the most time, and what parts could benefit from optimizations. In addition to the intervals of the above phases, three intervals are calculated:

1. The total collection cycle time.
2. Collector only time - the mark and sweep phases alone.
3. Mutator only time - the time between garbage collection cycles.

The total collection cycle time, that is the sum of the phases, gives us an indicator of how much overhead the garbage collector incurs. Throughout the stop the world phase, however, mutators can continue running, and only when the mark phase is reached is the garbage collector working alone. Because of this, the total may be too high, and *collector only time* is recorded as well. The intervals between collection cycles is important for mutators, as this is their time to perform work. Note that resumption is asynchronous by way of messaging, thus COGs will not resume instantly upon the time registered as the resumption time. Also during this time, tasks may send messages to the garbage collector to register new objects, futures and COGs, so the collector may still perform some work in this interval. Still, the *mutator only time* is a good indicator of how much time the mutators have to execute.

All time intervals are registered by the garbage collector, when garbage collection statistics logging is enabled. The I/O involved in logging, incurs some additional overhead for the collector, but is necessary for the evaluation. When measuring total execution time, however, the back end is compiled with this logging disabled. The execution time is then measured with Linux `time` tool, which measures overall execution time, time spent executing regular code and time spent on system calls.

It is important to note that the simulator shuts down by observing that the simulation has been idle for one second. Therefore the total execution time of a simulation will always exceed one second. The garbage collector may or may not continue to register some statistics early on in this one second idle time, but as mutator events will come to

a stop, the logging will stop shortly after. Therefore there may be a small discrepancy between the observations made by the garbage collector and the overall execution time, which will be longer. This means that comparing the total execution time of different simulations, with and without garbage collection, may have a larger relative difference than at first sight. The relative execution time is therefore given with and without idle time, i.e. deducting one second.

6.1.2 Memory Usage

The point of garbage collection is to make memory available for data without the programmer having to manually reclaim it. Therefore measuring memory usage throughout the program is essential. The measurements should happen regularly, and should work well also for relatively short-lived simulations. For long-lived simulations, it would suffice to use standard OS tools, but it would be hard to catch details of short-lived simulations. The reason for this is that most external tools have intervals that are longer, or require that the process ID is known before they can monitor the program. Starting monitoring after the simulator has started running, may lead to early data not being recorded. Longer intervals between measurements sacrifices accuracy for reduced noise. The garbage collector will already be logging collection events. The same logging capabilities can be used for registering memory usage, and the collector can produce a detailed view of the anatomy of the simulation in terms of objects, futures and COGs as well. Measuring it within the program would therefore be preferable.

Erlang makes information about dynamically allocated memory available through the `erlang:memory()` function. Some memory allocated by the virtual machine is not returned by this call, but it should give an adequate overview of the memory used in the simulation. “Shared libraries, the code of the emulator itself, and the emulator stack(s) are not supposed to be included. ... Furthermore, due to fragmentation and pre-reservation of memory areas, the size of the memory segments which contain the dynamically allocated memory blocks can be substantially larger than the total size of the dynamically allocated memory blocks” [6].

6.1.3 Processes

The limits set on the number of processes, makes it likely the simulation will run out of processes before memory unless the limit is increased. Particularly for models where objects and futures are numerous compared to their size, the process limit can be crippling. Therefore the number of processes is an important factor to measure. The number of processes created by the back end is directly related to the number of objects, futures and COGs. Until a future is resolved, there is also a process for the task whose result the future is to hold.

There are additional processes, which make up the underlying system of the back end, and the number of processes in total are made available by Erlang and registered in the log along with memory usage and the number of objects, futures and COGs.

6.1.4 Obtaining and Processing Measurements

Because operating systems typically reduce CPU clock speed and voltage to reduce power consumption, it is necessary to run many simulations in a loop to reduce variability in measurements. This can be done in two ways, either by running the code many times within a program, or running the program many times in a shell script. The former is good for measuring the speed of algorithms or expectations for a program when run on large data sets. While the latter gives a good indication of expected running time of a program overall, as any startup and termination costs are included. I have opted for the latter to measure average execution time, but will also show the time spent on a large loop variant of one of the test cases to show the overhead in a long-running simulation.

To get representable data for memory usage, process counts and GC events, multiple runs are also performed. The reason for doing multiple simulations is to avoid reporting statistics from particularly good or bad runs, i.e. avoiding bias. Each simulation is executed an odd number of times, and the statistics from the simulation with the median running time is chosen to be presented. The number of simulations is much lower when obtaining memory statistics than when finding average execution times, because of the amount of I/O operations for outputting the statistics, and then converting the log files to CSV for processing in R, takes up quite a long time.

The execution of simulations and finding the median result is scripted. The shell script in listing 6.1 is used for generating statistics and picking the median. The script is executed as follows:

```
./create_stats.sh Simulation experiment
```

where `Simulation` is the name of the model to simulate and `experiment` is the name of the garbage collection experiment. The two along with the number of the execution are used for the names of the output data sets. Finding the median runtime for a logged execution is done with an R script that finds the earliest and latest logged event, see listing 6.2. It is therefore based on the execution from the viewpoint of the garbage collector, and does not take VM startup and termination into account.

The script for measuring execution time is similar to the previous shell script, but simply runs the simulation without any logging or processing of logs, see listing 6.3. The script's entire execution is timed by executing it as follows

```
time ./runmany.sh Simulation 50
```

where the number after the model name is the number of times the simulation is executed.

Listing 6.1: Script to produce and choose statistics from 11 simulations.

```
#!/bin/bash

for i in {1..11};
do
    ./run -g $1 > $1-$2$i.txt
    ./gcstats_as_csv.erl $1-$2$i.txt
done

./find_median_runtime.R $1-$2
```

Listing 6.2: A simple R script that finds the data set with the median runtime.

```
#!/usr/bin/env Rscript

runtimes <- c(0)
prefix = commandArgs(TRUE)[1]
i <- 1
while (file.exists(
    paste(prefix, i, "-mem.csv",sep=""))) {
  times <- read.csv(
      paste(prefix,i, "-mem.csv",sep=""))[,1]
  times <- append(times, read.csv(
      paste(prefix, i, "-events.csv", sep=""))[,1])
  times <- append(times, read.csv(
      paste(prefix, i, "-sweeps.csv", sep=""))[,1])
  runtimes[i] <- max(times) - min(times)
  i <- i + 1
}

print(order(runtimes)[round(length(runtimes) / 2)])
```

Listing 6.3: Script that executes a simulation many times.

```
#!/bin/bash

for i in `seq 1 $2`;
do
    ./run $1
done
```

The log files are then processed with another R script which generates a plot of memory usage, the number of objects, COGs and futures. It calculates the intervals from the timestamps in the log.

The data is then processed into summaries and output into tables with minimum, maximum, mean and median values for intervals, sweeps and the plotted data along with the number of processes. This gives a good overview of an execution, and what work the garbage collector has performed. The minimum values are mostly of interest for the intervals, as they are almost always zero for other reported values like the number of objects. Therefore the minimum is removed from the other tables in this report.

All experiments were run on workstations at the University of Oslo with the following software and hardware:

Operating System Red Hat Enterprise Linux Workstation release 6.6

CPU Intel Core i7 CPU 870 2.93 GHz

RAM 2 × 4 GB DDR3 1333 MHz

Erlang Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false]

6.2 Test Cases

There are four test cases that have been run. The purpose of the numerous test cases is to test how well the garbage collector deals with different types of programs. Below I present each of the models, and why they have been used.

6.2.1 Ping Pong - Basic Test with Cyclic Garbage

This is the most basic test case, and is similar to a "Hello, world!" program, but illustrates message passing. It is one of the example programs in the ABS distribution, and comes in two flavors. The first simply has two objects, `Ping` and `Pong`, which exchange a set of messages: `hello`, `how are you`, `fine`, `bye` and `bye`. The number of objects and futures involved in the simple flavor, is so low, it does not really produce any interesting results. The second flavor establishes sessions within the `Pong` object, for exchanging messages with a number of `Ping` objects. I will present the results of running the second flavor, creating many Ping objects, which will be able to show the garbage collector working.

The Ping Pong example is also a good test case for garbage collection, because it is one of the simplest examples of cyclic garbage. `Ping` objects reference `PongSession` objects, which reference their respective `Ping` objects. When they finish executing, they should be collectible, if the garbage collector can identify cyclic garbage. We will see that this is the case in the experiments performed.

For the basic run, that is used for comparison of running times. I have used a loop that creates one hundred `Ping` objects. There will also be two futures for each session throughout the communication,

one for the `Ping` object's calls to the `PongSession`, and one for the `PongSession` object's calls to the `Ping` object. The `Ping` object waits for its calls to finish, but it does not finish until the session has made a call to the `Ping` object. See listing 6.4 for an extract of the code, showing the communication between the `Ping` object and its session.

Listing 6.4: Extract from the MultiPingPong example.

```
class PingImpl(Pong pong) implements Ping {
    PongSession pongSession;
    Unit run(){
      Fut<PongSession> fu = pong!hello(this);
      pongSession = fu.get;
    }

    Unit ping(PingMsg msg){
      PongMsg reply = case msg {
         HelloPing => HowAreYou;
         Fine => ByePong;
         ByePing => NoMsg;
      };

      if (reply != NoMsg) {
         Fut<Unit> fu = pongSession!pong(reply);
         fu.get;
      }
     }
}

class PongSessionImpl(Ping ping, [Near] PongIntern pong)
        implements PongSession {
    // init block
    {
       ping!ping(HelloPing);
    }

    Unit pong(PongMsg msg){
       if (msg == HowAreYou) {
           ping!ping(Fine);
       } else {
           ping!ping(ByePing);
           pong.sessionFinished(this);
       }
    }
}
```

Additionally, a run with many more objects is made, to show that the garbage collector enables continuous operations. By setting it to create `Ping` objects infinitely, a simulation with the simulator before garbage collection was implemented, crashes after a few seconds. It takes much longer to crash with the garbage collector running with collection turned off. This comes from the synchronous creation of COGs and futures, which is the main work performed in the program. Logging also slows the entire simulation down considerably.

Because a garbage collected version could run for a very long time, I

set up the script to automatically kill the simulation after one minute. Running the simulation much longer, would create very large log files. Each second of runtime yields approximately 1 MiB of logged data for this particular test on the workstation used. The purpose of the test is just to show that the garbage collector works, and for this, the one minute run is sufficient. However, there are some issues with the collector that this test highlights, but due to time, adequate solutions have not been found for these issues.

6.2.2 Sequences - Asynchronous Method Calls in Loops

To show the use of being able to stop running tasks, a simple model was constructed that calls methods asynchronously in a loop without synchronization points. The example consists of objects that generate a value in a sequence, when they receive a method call. This example can be seen as an implementation of a lazily evaluated generator, which can be used to implement streams in ABS. Streams are a kind of list where the elements are generated only when they are needed, and may be infinite [1].

The extract in listing 6.5 illustrates how this test case works. A loop runs for several iterations without yielding, and makes asynchronous calls on an object on another COG. This creates many futures in quick succession. Should the garbage collector attempt to stop the world, the loop main COG will not stop until the loop has completed. This leads to a long pause, where the `Sequence` object's COG is paused and doesn't allow any work to be performed, while the main task continues to produce more work. The full version can be found in Appendix B, which has multiple types of `Sequence` objects that receive asynchronous calls, yielding a slightly higher load than this example.

Listing 6.5: Extract from the Sequences test case.

```
class NaturalNumbers implements Sequence {
    Int i = 0;

    Int next() {
        i = i + 1;
        return i;
    }
}

{
    Int i = 0;
    Sequences s = new Fibonacci();
    while (i < 1000) {
        s ! next();
        i = i + 1;
    }
}
```

6.2.3 Prime Sieve - Long-running Tasks

An example of long-running tasks is needed to show the usefulness of using a timed trigger. First of all long-running tasks should be left alone, to do their work as long as there is no other reason to collect garbage. Triggering by counting objects, would achieve this, but it would likely not react to tasks completing after a long time, because the count would be likely to not reach the threshold. A timed trigger, would detect that garbage collection is overdue when a task is complete and the associated future stops being a root. Thus it would trigger garbage collection whether there has been a significant rise in processes.

The chosen example is a simple number crunching program, that calculates prime numbers. It checks primality, by dividing the number by all primes lower than it. To avoid long insertion times of new primes, it adds them in reverse order. This also makes it likely to take longer to calculate that a number is not a prime number, because it is more likely to be divisible by a lower prime, than a higher prime. Thus even if fast insertions are used, the sieve is rather slow, leading to long-running tasks.

6.2.4 Indexing - Resolved Futures Held

The last test consist of an example of MapReduce, used as motivating examples for variability modeling in ABS [9]. In the paper they used the variability modeling capabilities of ABS to produce different versions of MapReduce examples. The deployments differed in terms of number of computers, the computers' capacity to perform work and the cost of jobs. Variability modeling has not been covered in this thesis, but the example models from the paper are still useful. However, during the simulations, no limits were set on the number of workers or capacity of workers, i.e. no changes were made through the use of ABS's variability modeling capabilities.

MapReduce performs work over large key value data sets across a number of computers. When the work is complete, the results are treated in parallel over a number of computers to combine the results into a single result. The two tasks performed, performing work over a data set and combining the results, are called map and reduce respectively. A full description of MapReduce can be found in [4].

Unfortunately, very little garbage is produced by the MapReduce examples, because the model holds on to the results until they have been reduced. It is a good test to see if the garbage collector does any incorrect collection, as it produces a lot of objects and slightly more complex tasks than the other test cases. It also holds on to futures that must not be collected, even if they have been resolved. Some collection is still possible towards the end of the simulation, but then the processes are going to be stopped along with the simulation ending anyway.

The Indexing example consist of a model of a part of a search engine. In this case, it is an example of indexing documents based on occurences

of sub-strings, generating what is known as an inverted index.

6.3 Results

The experiments performed, used the different triggers mentioned in section 5.6. In addition, a select few were tested both with the ability to stop running tasks and without. I will present the results by test case. Because of the large amount of data collected, all the results can be found in the appendices, while a selection has been made for this presentation here.

6.3.1 Ping Pong - Results

Let us first have a look at the overall runtimes for the *Ping Pong* case. To be able to fit the table, I have not included user and system time, which reveal very little about the garbage collector anyway. The last column, *Relative without idle*, contains the time relative to the baseline, i.e. before garbage collection was added, after deducting one second of the time. This second is spent by the system to ensure that there are no tasks on the air, waiting to be added to COGs. After the second has passed it shuts down the simulator.

Table 6.1: Ping Pong - Average runtimes for 50 runs

Collection scheme	Real time	Relative	Relative without idle
Before GC	1274 ms	100.0 %	100.0 %
Never collect	1257 ms	98.6 %	93.5 %
Always collect	1854 ms	145.5 %	311.3 %
Collect on count	1293 ms	101.4 %	106.7 %
Collect on time	1261 ms	98.9 %	94.9 %
+ Stop running	1265 ms	99.2 %	96.5 %
Collect on either	1290 ms	101.2 %	105.5 %
+ Stop running	1298 ms	101.8 %	108.4 %

In some of the cases in Table 6.1, runtimes have decreased from the baseline. This tells us there is at least some noise in the samples, even if they have been run multiple times and averaged. Therefore we can assume that measured times may vary by around 10 ms in either direction, possibly more. The one thing we can deduce from the numbers, is that collecting on every event is not a viable choice. The other runs stay within the noise range.

In Figure 6.1 we can see what the progression of a simulation looks like, if the trigger is set to never initiate garbage collection. As is expected, the number of objects, futures and COGs all rise throughout this program, as does memory usage. Have a look at Appendix A if you want to compare the results of more triggers.

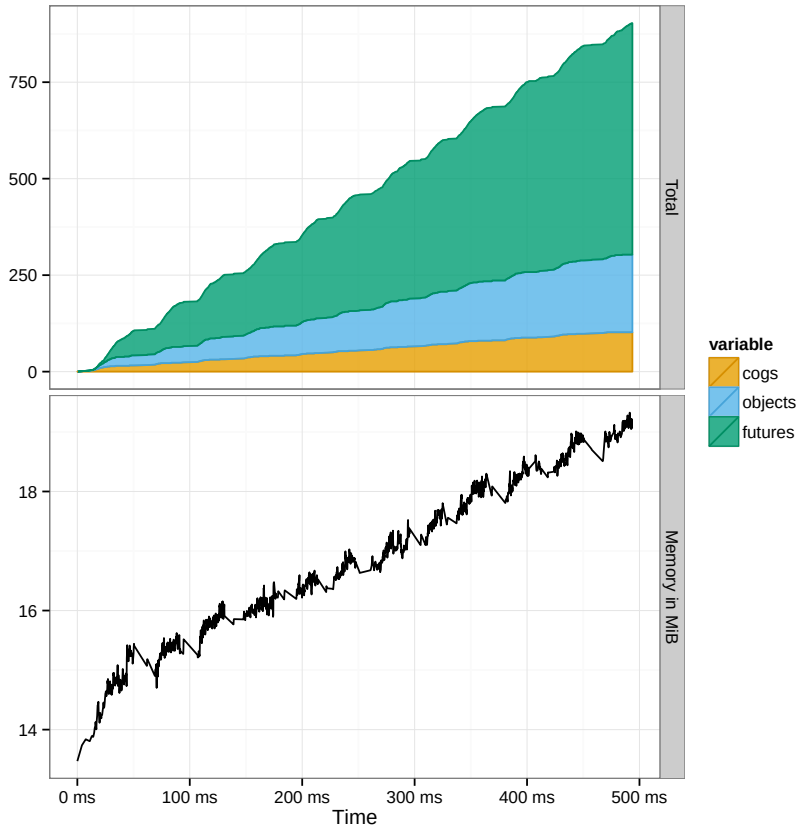

Figure 6.1: Ping Pong - No collection - Memory and counts plot

In the case of the *Ping Pong* example, being able to stop running tasks does not do much for the garbage collector. Therefore the counting trigger and timed trigger are presented instead in Figure 6.2 and Figure 6.3 respectively. We can see that the counting trigger is more aggressive than the timed trigger. The spikes in futures, tells us that many futures are regularly collected, but new ones are soon allocated again, triggering a new collection. Because of the eager behavior brought on by this trigger, there's not much reason to look at the combined timer and counter. The counter would collect so eagerly, the timer would never be needed to trigger collection. We will see this across most of the experiments that have been run.

The timed trigger had slightly faster runtimes than the counting trigger, but the counting trigger clearly has lower object counts. Memory-wise they are only about 1 MiB apart. There is no obvious cadidate just from looking at these data. Comparing the interval times, objects and futures swept per cycle, and amount of memory and objects used, does not help much either, these are found in Appendix A. As we have already seen, the timed collector comes out on top in terms of time spent, while the counting collector comes out on top in memory usage and process counts.

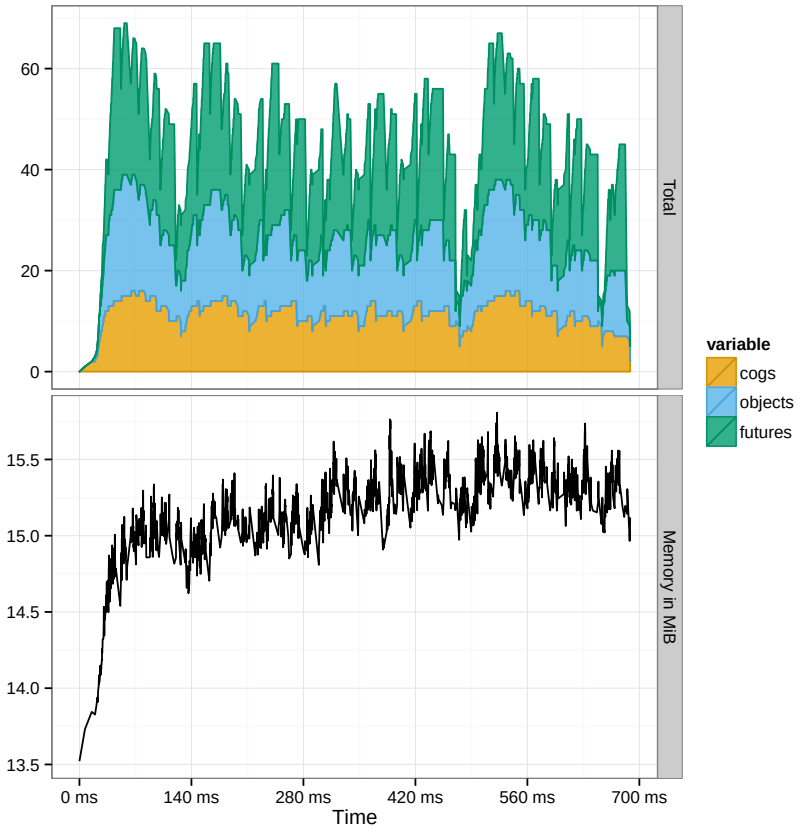

Figure 6.2: Ping Pong - Trigger on count - Memory and counts plot

Infinite loops

When changing the loop to an infinite loop, instead of the hundred iterations in the basic test. The garbage collector doesn't cope quite as well with this test case. The back end as it was before the garbage collector implementation, only executes for a few seconds before crashing. However, with the garbage collector implemented, but with garbage collection turned off, it takes much longer, as seen in Figure 6.4.

This could be due to a combination of many things: the logger takes up a lot of time and resources, large sets of messages are being passed around, and a close inspection of the raw data shows that steadily the number of futures that are resolved declines. This means futures cannot be collected as quickly, which impacts the collection of objects, which are waiting for the futures to be resolved. This could be a sign that the overhead on tasks is too high, leading to slower execution when the number of processes becomes very large. The problems with a growing root future set can be seen in Figure 6.5.

However, this test case is designed to be harder to cope with than most real models. It will be shown in subsection 6.3.4 that it can cope well with a large real world model.

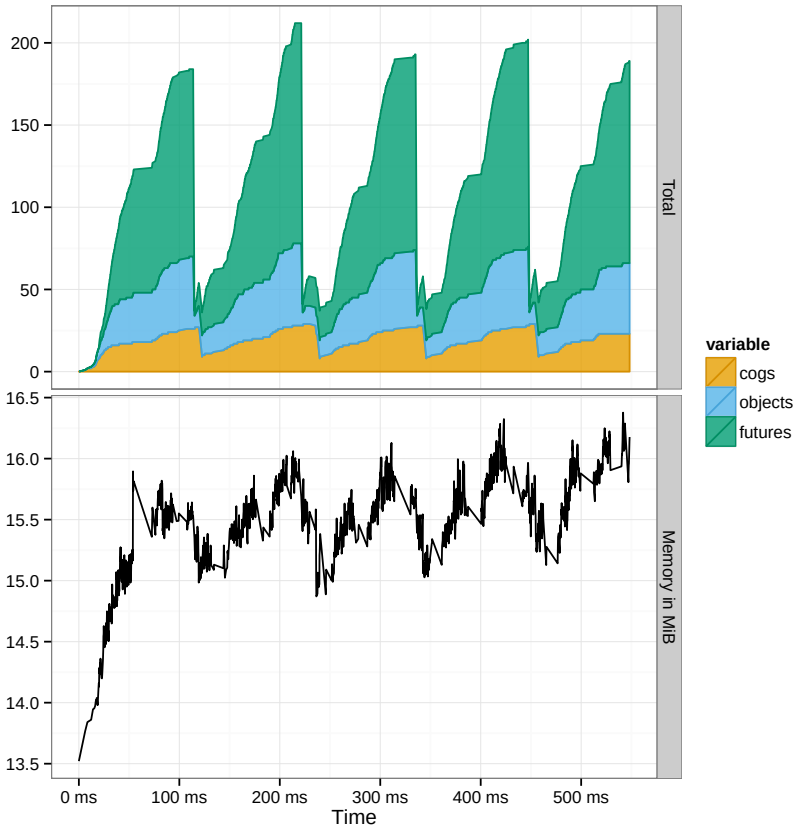

Figure 6.3: Ping Pong - Timed trigger - Memory and counts plot

6.3.2 Sequences - Results

The sequences example was specifically designed to show the need for being able to stop running tasks. First, let us look at the overall runtimes in Table 6.2. We can see that garbage collection comes out much worse in this example. Just enabling the mutator changes and record-keeping parts of the garbage collector leads to an increase of over 10%, after deducting idle time. The counting trigger does not do very well at all in this particular test, taking about the very least nine times as long. The timed collector does much better, and enabling stopping running tasks, makes it even better, although the time increase is still significant.

For comparison purposes, I present the plots for the timed collectors, with and without stopping running tasks in Figure 6.7 and Figure 6.6 respectively. The timeline indicates that logging severely impacts the garbage collector. The graph spans about three times as long an interval compared to the runtimes recorded, even with the graph not showing the idle time at the end.

The first collection made by the timed collector without stopping tasks, takes a very long time. This can be seen as a very long straight line from around the one second mark, until just before the 2.5 second

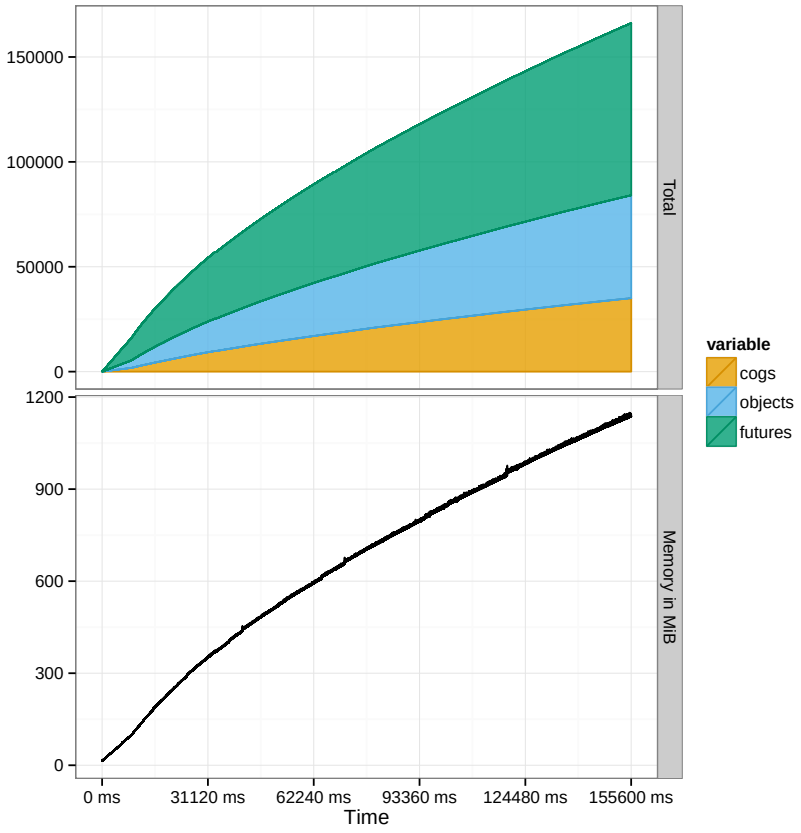

Figure 6.4: Infinite Ping Pong - No collection - Memory and counts plot

mark. Very little collection occurs until the second and third collection cycles, which are also delayed longer than the regular 100 ms. This may be due to a lack of events in parts of the timespan, with fewer events, less changes are seen in the memory usage graph.

With the ability to stop running tasks, it fares much better. It collects garbage regularly and the pauses are much shorter. Memory usage has also been halved in this result. Looking at the length of intervals in the garbage collection cycles, Table 6.3 and Table 6.4, also show that stopping the world and marking both take only about 10% of the time after adding the ability to stop the world. The garbage collector runs many more cycles, but increasing the timeout would likely bring down execution time even more.

6.3.3 Prime Sieve - Results

The prime sieve test case was designed to see what happens when processes that take a long time to execute are added to a model. The results here are quite interesting, both counting and timed triggers have approximately the same runtimes as is shown in Table 6.5. However, adding the ability to stop running tasks, does not improve the execution time in this case. Instead it slows down the computation of prime

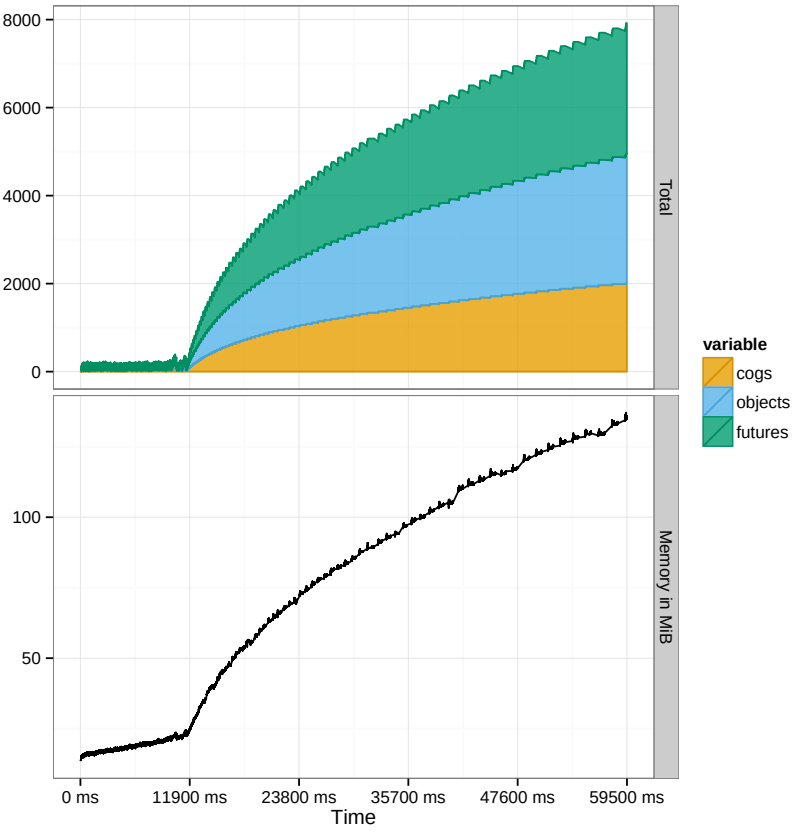

Figure 6.5: Infinite Ping Pong - Timed trigger - Memory and counts plot

numbers because the tasks are interrupted. These tasks do not produce new tasks, which would benefit from stopping as was seen in the *sequences* test. As more resources are freed, new tasks would be allowed to run unhindered.

There is one benefit to stopping the tasks while they run, even if it leads to slightly longer execution times. The collectors that do not do so, have to wait for all the tasks to finish, which makes for the entire duration of the execution. The versions that stop tasks, are able to collect garbage during the collection. This also results in a much shorter *stop the world* phase, see Table 6.7.

6.3.4 Indexing - Results

The MapReduce test case keeps references to futures until they have been reduced. This results in almost no garbage being collected. The different triggers, come out with similar results, except when always triggering events. The runtimes are shown in Table 6.8.

The most interesting part, is therefore the time spent in the garbage collector. The counting trigger comes out with the shortest amount of time spent in the garbage collector overall. Being able to stop running tasks, is not necessary in this case, as there's mostly a large amount of

Table 6.2: Sequences - Average runtimes for 30 runs

Collection scheme	Real time	Relative	Relative without idle
Before GC	1395 ms	100.0 %	100.0 %
Never collect	1439 ms	103.2 %	111.2 %
Always collect	39159 ms	2807.2 %	9662.1 %
Collect on count	16801 ms	1204.4 %	4001.0 %
Collect on time	1622 ms	116.3 %	157.5 %
+ Stop running	1530 ms	109.7 %	134.1 %
Collect on either	17054 ms	1222.6 %	4065.1 %
+ Stop running	12614 ms	904.3 %	2940.8 %

Table 6.3: Sequences - Timeouts - Intervals

	Mean	Median	Max	Total
Stop world	582425 µs	648370 µs	931294 µs	2329700 µs
Mark	364064 µs	26022 µs	1404118 µs	1456258 µs
Sweep	2632 µs	2304 µs	5169 µs	10530 µs
Collection cycle	949122 µs	678765 µs	2336164 µs	3796488 µs
Collector only	366697 µs	30394 µs	1404870 µs	1466788 µs
Mutator only	100543 µs	100312 µs	101469 µs	402171 µs

short-lived tasks to stop.

Table 6.4: Sequences - Timeouts with stopping - Intervals

	Mean	Median	Max	Total
Stop world	54606 µs	42809 µs	182148 µs	1146725 µs
Mark	7033 µs	5109 µs	22563 µs	147699 µs
Sweep	457 µs	415 µs	926 µs	9601 µs
Collection cycle	62096 µs	53257 µs	183463 µs	1304025 µs
Collector only	7490 µs	5849 µs	22775 µs	157300 µs
Mutator only	101705 µs	100169 µs	132022 µs	2135811 µs

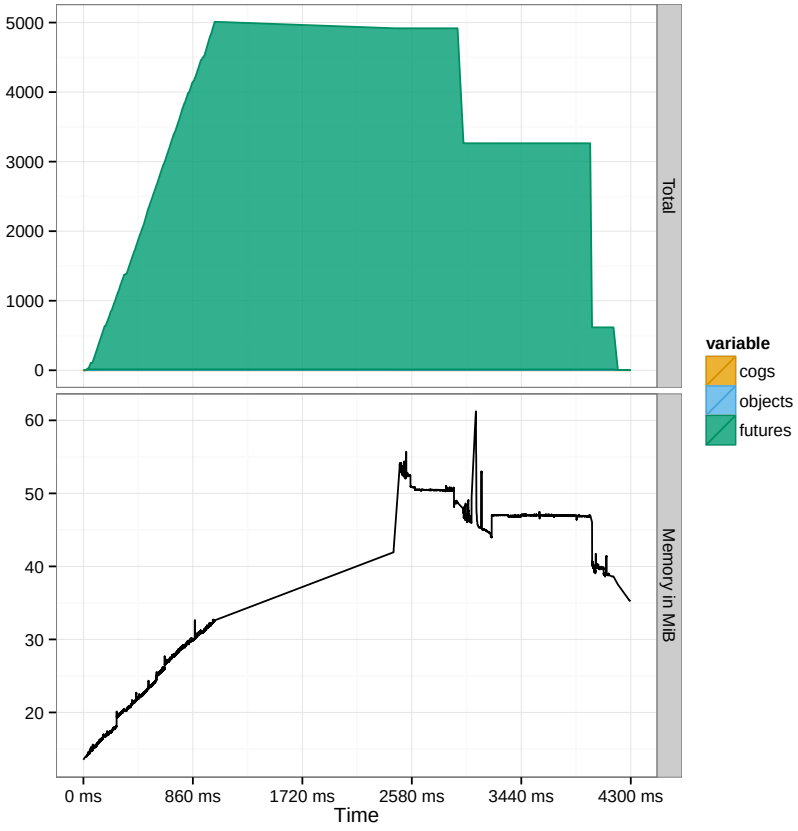

Figure 6.6: Sequences - Timeouts - Memory and counts plot

Table 6.5: Prime Sieve - Average runtimes for 50 runs

Collection scheme	Real time	Relative
Before GC	1448 ms	100.0 %
Never collect	1453 ms	100.3 %
Always collect	1903 ms	131.4 %
Collect on count	1482 ms	102.4 %
Collect on time	1454 ms	100.4 %
+ Stop running	1542 ms	106.5 %
Collect on either	1472 ms	101.6 %
+ Stop running	1545 ms	106.7 %

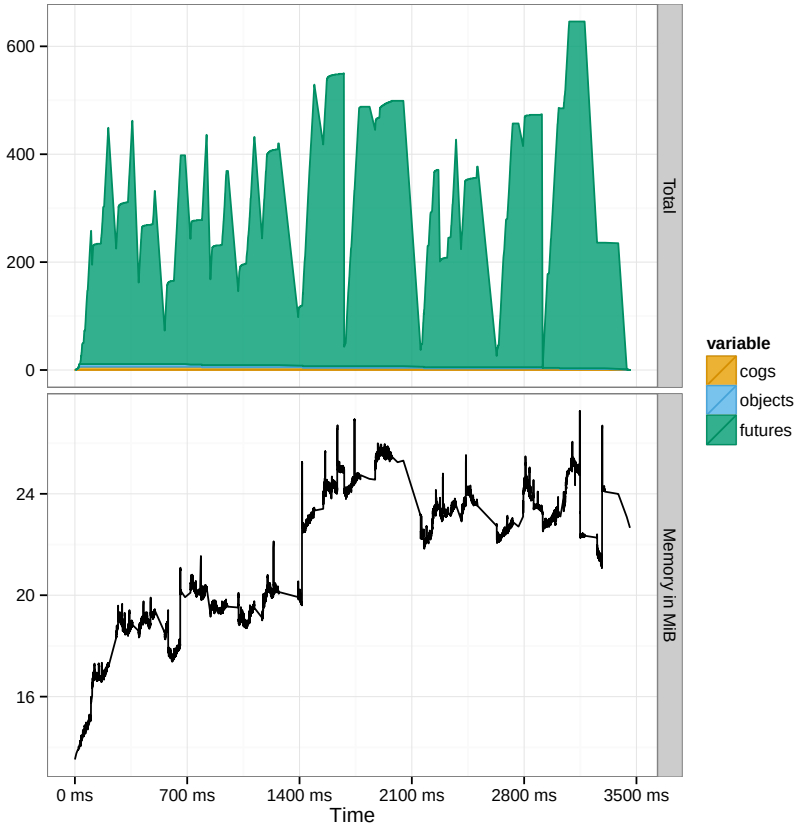

Figure 6.7: Sequences - Timeouts with stopping - Memory and counts plot

Table 6.6: Prime Sieve - Timeouts - Intervals

	Mean	Median	Max	Total
Stop world	222588 µs	222588 µs	222588 µs	222588 µs
Mark	272 µs	272 µs	272 µs	272 µs
Sweep	151 µs	151 µs	151 µs	151 µs
Collection cycle	223011 µs	223011 µs	223011 µs	223011 µs
Collector only	423 µs	423 µs	423 µs	423 µs
Mutator only	113730 µs	113730 µs	113730 µs	113730 µs

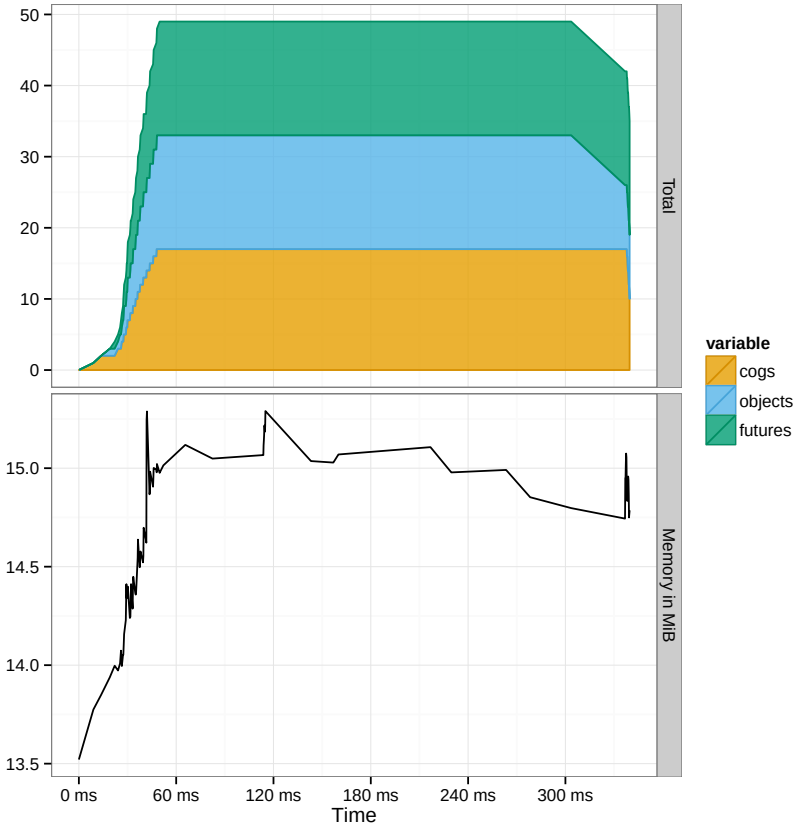

Figure 6.8: Prime Sieve - Timeouts - Memory and counts plot

Table 6.7: Prime Sieve - Timeouts with stopping - Intervals

	Mean	Median	Max	Total
Stop world	2055 µs	1773 µs	2881 µs	6165 µs
Mark	235 µs	197 µs	344 µs	704 µs
Sweep	100 µs	88 µs	148 µs	301 µs
Collection cycle	2390 µs	2035 µs	3373 µs	7170 µs
Collector only	335 µs	262 µs	492 µs	1005 µs
Mutator only	131047 µs	139763 µs	152415 µs	393142 µs

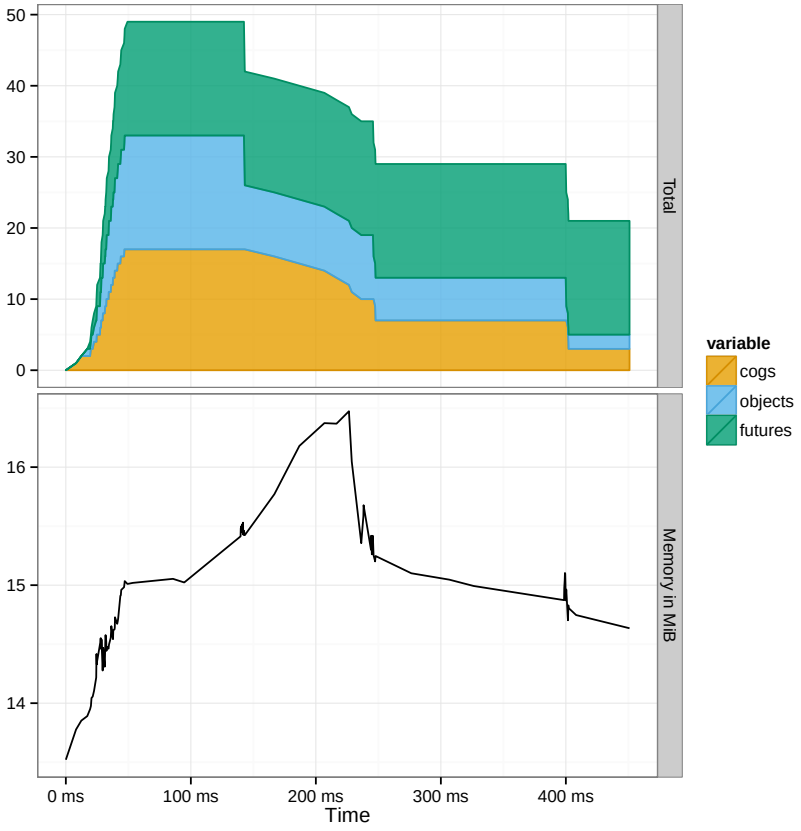

Figure 6.9: Prime Sieve - Timeouts with stopping - Memory and counts plot

Table 6.8: Indexing - Average runtimes for 50 runs

Collection scheme	Real time	Relative	Relative without idle
Before GC	1356 ms	100.0 %	100.0 %
Never collect	1373 ms	101.2 %	104.7 %
Always collect	5043 ms	371.8 %	1134.2 %
Collect on count	1408 ms	103.9 %	114.7 %
Collect on time	1398 ms	103.1 %	111.7 %
+ Stop running	1405 ms	103.6 %	113.9 %
Collect on either	1410 ms	104.0 %	115.3 %
+ Stop running	1434 ms	105.7 %	121.9 %

Table 6.9: Indexing - Counting trigger - Intervals

	Mean	Median	Max	Total
Stop world	14376 µs	6106 µs	54589 µs	100632 µs
Mark	3298 µs	1161 µs	13353 µs	23084 µs
Sweep	175 µs	87 µs	572 µs	1227 µs
Collection cycle	17849 µs	7354 µs	68514 µs	124943 µs
Collector only	3473 µs	1248 µs	13925 µs	24311 µs
Mutator only	76180 µs	60382 µs	239159 µs	533257 µs

Chapter 7

Conclusion

In this thesis a garbage collector for the Erlang back end of ABS has been described and developed. It has been designed according to the criteria set in the introduction, speed, completeness and correctness:

- It should not invoke considerable slowdowns to the system.
- It should collect all or, at the very least, most identifiable garbage.
- It should not collect objects that may be needed in the future.

In chapter 5, the necessary steps to ensure the collector correct and complete were taken. While in chapter 6, we saw that collecting based on timeouts gave satisfactory results in general. It was able to keep memory usage and the number of processes in acceptable ranges overall, and usually it performed better than the other garbage collection triggers. In some cases, stopping running tasks was necessary to get acceptable collection results and avoid long pauses, althoughwhile in other cases the running time increased by a small amount.

The infinite loop version of the *Ping Pong* test, did display some issues with the garbage collector. The exact reasons for this is not easily determined, but some factors indicated it was due to overhead placed on mutators. This could make the garbage collector problematic in use, but further experimentation is required. Preferrably with real models, such as the *Indexing* test case, which would apply realistic loads, instead of simple tight loops leading to large amounts of objects and futures being created.

In the *Indexing* test case, which was taken from an Envisage case study, the garbage collector did not invoke any considerable slowdowns. This test case has a high load for real models with a large amount of parallel processes. Although the model creates many processes, the garbage collector is able to stop and resume the world in a short amount of time. This indicates it is likely to work well for the kinds of models normally simulated with ABS.

7.1 Future Work

Some of the unexplored areas in this thesis, are the implementation of concurrent collection and the consequences of developing a distributed version of the Erlang back end. When the Erlang back end eventually is made to run simulations over multiple nodes, latency will negatively impact the garbage collector, as it needs to communicate with all the nodes involved. A distributed version of the collector, where one collector runs on each node or for each COG, could be of interest to develop. Such a collector may be able to lower the number of messages between nodes, by grouping such messages into fewer slightly larger messages.

Concurrent collection would minimize the time spent exclusively by the garbage collector, but could require added locking mechanisms to avoid collecting objects prematurely. The active object collector developed in {caromel}, might be a good fit for ABS. How its use of additional active objects to ensure references in passive objects are seen, and its applicability to the Erlang back end would require further investigation. A distributed, concurrent mark and sweep, like the one used in Emerald [12], might prove a better fit.

The above experiments may be able to cope better with extreme cases, like the infinitely looping *Ping Pong* test case. Changing marking to an asynchronous model, possibly delegated to the other processes instead of the collector itself, might also help and shares similarities with the active object collector mentioned. If mutator overhead could be lowered, more tasks are likely to finish, allowing collection of more futures. This would require further investigation of the guarantees made to ensure correct collection.

Appendix A

Ping Pong Test Case

A.1 Source Code

```
/*
 * An extended PingPong example
 * where Pong can handle multiple Pings
 */
module PingPong;

data PingMsg = Fine
             | HelloPing
             | ByePing
             ;

data PongMsg = NoMsg
         | Hello(Ping)
         | HowAreYou
         | ByePong
         ;

interface Ping {
    Unit ping(PingMsg m);
}

interface Pong {
    PongSession hello(Ping ping);
}

interface PongSession {
    Unit pong(PongMsg m);
}

class PingImpl(Pong pong) implements Ping {
    PongSession pongSession;
    Unit run(){
      Fut<PongSession> fu = pong!hello(this);
      pongSession = fu.get;
    }

    Unit ping(PingMsg msg){
      PongMsg reply = case msg {
         HelloPing => HowAreYou;
```

```
         Fine => ByePong;
         ByePing => NoMsg;
      };

      if (reply != NoMsg) {
         Fut<Unit> fu = pongSession!pong(reply);
         fu.get;
      }
     }
}

class PongSessionImpl(Ping ping, [Near] PongIntern pong)
        implements PongSession {
    // init block
    {
       ping!ping(HelloPing);
    }

    Unit pong(PongMsg msg){
       if (msg == HowAreYou) {
           ping!ping(Fine);
       } else {
           ping!ping(ByePing);
           pong.sessionFinished(this);
       }
    }
}

interface PongIntern {
    Unit sessionFinished(PongSession s);
}

class PongImpl implements Pong, PongIntern {
    List<[Near] PongSession> sessions = Nil;
    PongSession hello(Ping ping) {
       PongSession s = new local PongSessionImpl(ping,this);
       sessions = appendright(sessions,s);
       return s;
    }

    Unit sessionFinished(PongSession s) {
        sessions = without(sessions,s);
    }
}

{
    Pong pong = new PongImpl();
    Int i = 0;
    while (i < 100) {
        new PingImpl(pong);
        i = i + 1;
    }
}
```

A.2 Results

A.2.1 Never Triggering Collection

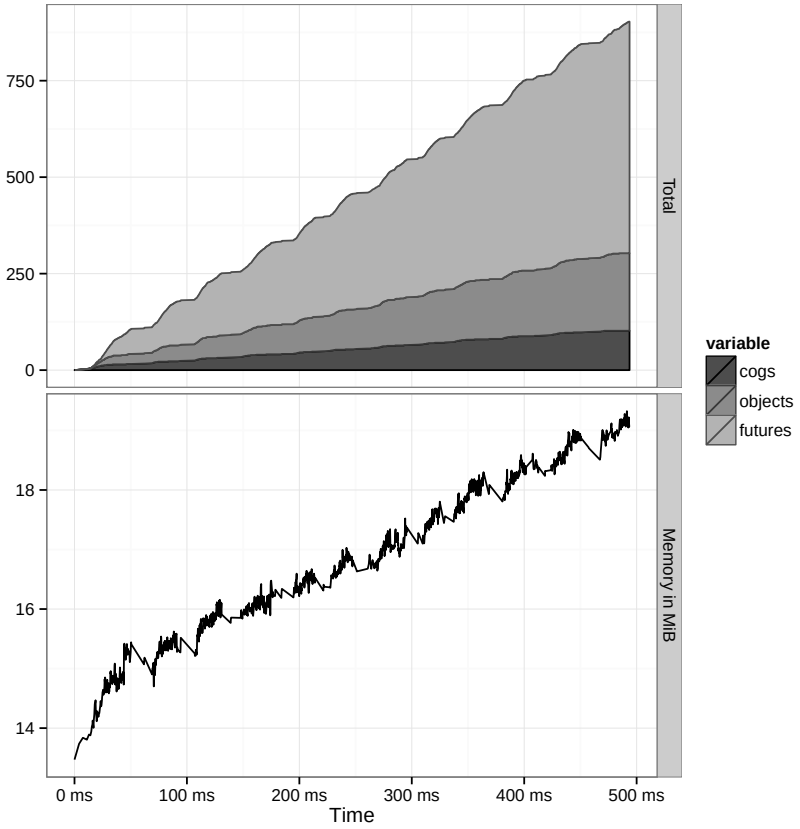

Table A.1: Ping Pong - Never Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	16.9 MiB	16.8 MiB	19.3 MiB
COGs	0	56.9	54	102
Objects	0	106.6	103	201
Futures	0	289.3	295.5	600
Root futures	0	9.4	9	20
Processes	24	560.4	544	1037
Φ	0.9 ‰	21.4 ‰	20.8 ‰	39.6 ‰

A.2.2 Always Triggering Collection

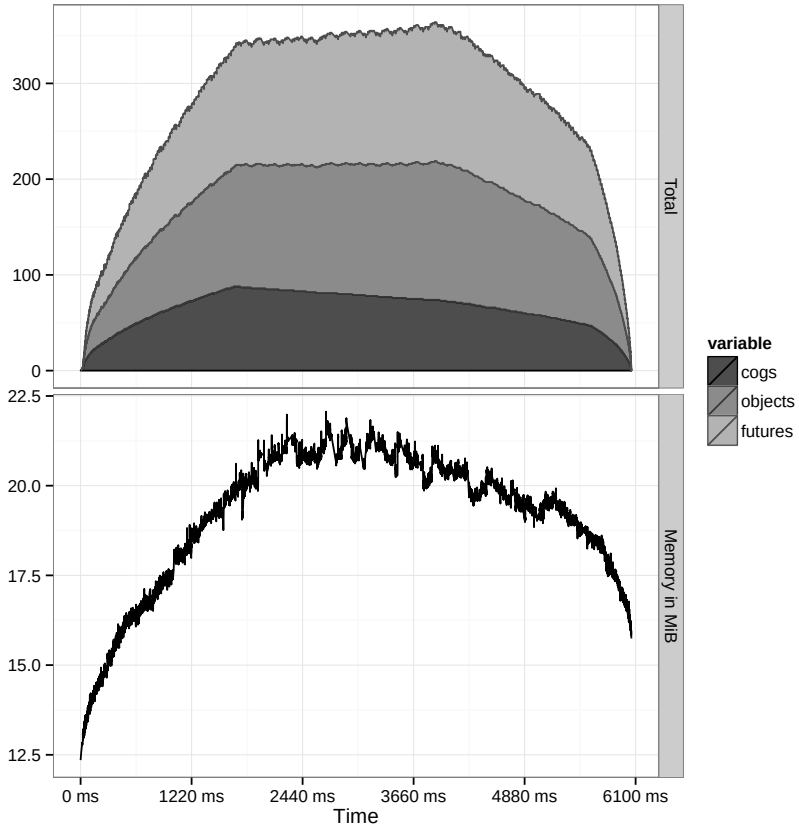

Table A.2: Ping Pong - Always Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	12.4 MiB	19.2 MiB	19.7 MiB	22.1 MiB
COGs	0	65.3	71	88
Objects	0	110.3	125	145
Futures	0	1.9	2	5
Root futures	0	108.7	123	144
Processes	24	529.7	588	645
Φ	0.9 ‰	20.2 ‰	22.4 ‰	24.6 ‰

Table A.3: Ping Pong - Always Collect - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.7	0	3
Objects kept	0	93.6	107	144
Futures swept	0	1.9	2	5
Futures kept	0	0.8	1	1

Table A.4: Ping Pong - Always Collect - Intervals

	Mean	Median	Max	Total
Stop world	16444 µs	15396 µs	57757 µs	5064852 µs
Mark	2778 µs	3092 µs	11305 µs	855609 µs
Sweep	113 µs	116 µs	265 µs	34898 µs
Collection cycle	19336 µs	19014 µs	63027 µs	5955359 µs
Collector only	2891 µs	3238 µs	11411 µs	890507 µs
Mutator only	318 µs	248 µs	7482 µs	97798 µs

A.2.3 Trigger on Timeout

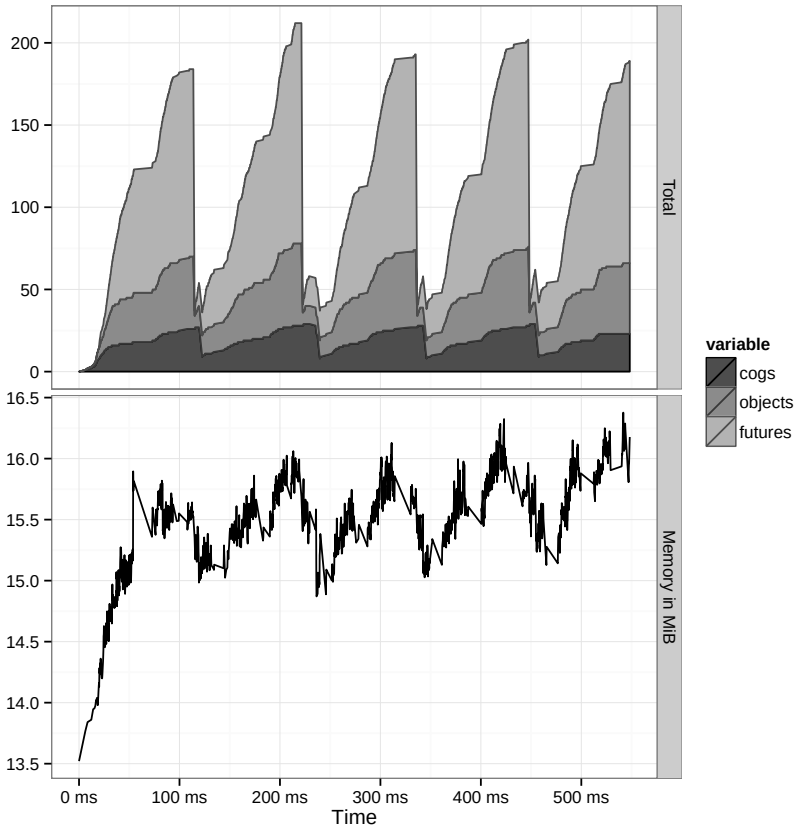

Table A.5: Ping Pong - Collect on time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.5 MiB	15.6 MiB	16.4 MiB
COGs	0	20.2	20	29
Objects	0	29.4	28	50
Futures	0	57.2	57	130
Root futures	0	8.2	8	24
Processes	24	210.8	209.5	321
Φ	0.9 ‰	8 ‰	8 ‰	12.2 ‰

Table A.6: Ping Pong - Collect on time - Sweeps

	Min	Mean	Median	Max
Objects swept	36	39.5	40	42
Objects kept	7	7.8	8	8
Futures swept	111	119.2	118.5	129
Futures kept	0	0.2	0	1

Table A.7: Ping Pong - Collect on time - Intervals

	Mean	Median	Max	Total
Stop world	5998 µs	6093 µs	6498 µs	23992 µs
Mark	330 µs	328 µs	340 µs	1319 µs
Sweep	750 µs	747 µs	813 µs	2998 µs
Collection cycle	7077 µs	7139 µs	7570 µs	28309 µs
Collector only	1079 µs	1070 µs	1153 µs	4317 µs
Mutator only	105024 µs	106213 µs	107620 µs	420098 µs

A.2.4 Trigger on Timeout with Stopping Running Tasks

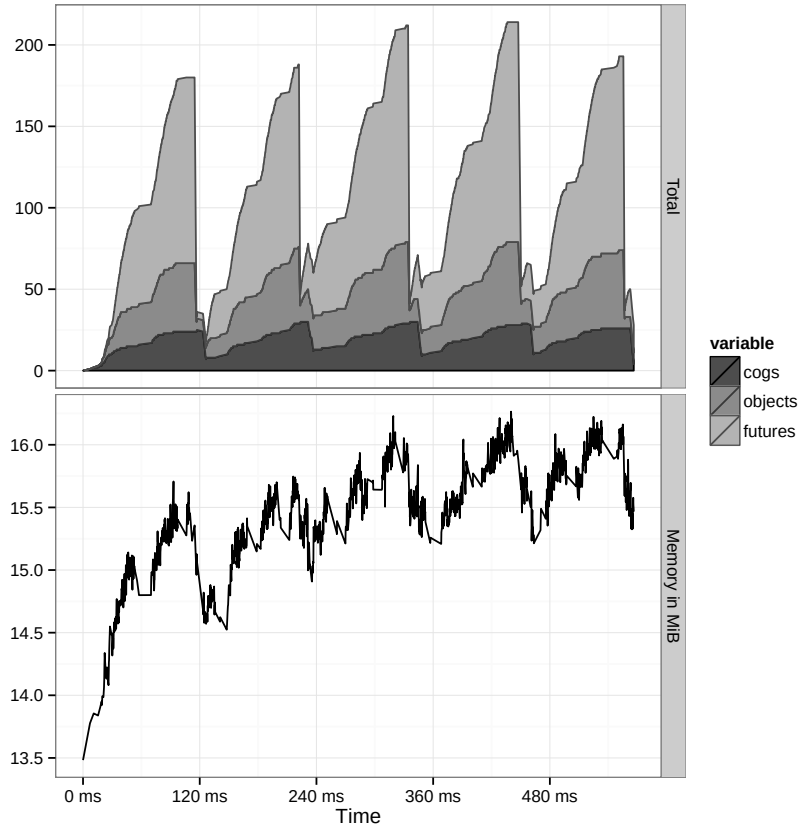

Table A.8: Ping Pong - Collect on time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.5 MiB	15.5 MiB	16.3 MiB
COGs	0	20.4	21	30
Objects	0	29.4	29	51
Futures	0	56.7	57	129
Root futures	0	8.6	8	20
Processes	24	212	216	338
Φ	0.9 ‰	8.1 ‰	8.2 ‰	12.9 ‰

Table A.9: Ping Pong - Collect on time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	36	38.8	38	42
Objects kept	6	8.8	8	13
Futures swept	105	116.6	114	129
Futures kept	0	0	0	0

Table A.10: Ping Pong - Collect on time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	7721 µs	5522 µs	13227 µs	38605 µs
Mark	333 µs	323 µs	422 µs	1667 µs
Sweep	755 µs	747 µs	905 µs	3776 µs
Collection cycle	8810 µs	6516 µs	14461 µs	44048 µs
Collector only	1089 µs	1021 µs	1250 µs	5443 µs
Mutator only	102550 µs	100300 µs	106219 µs	512750 µs

A.2.5 Trigger on Count

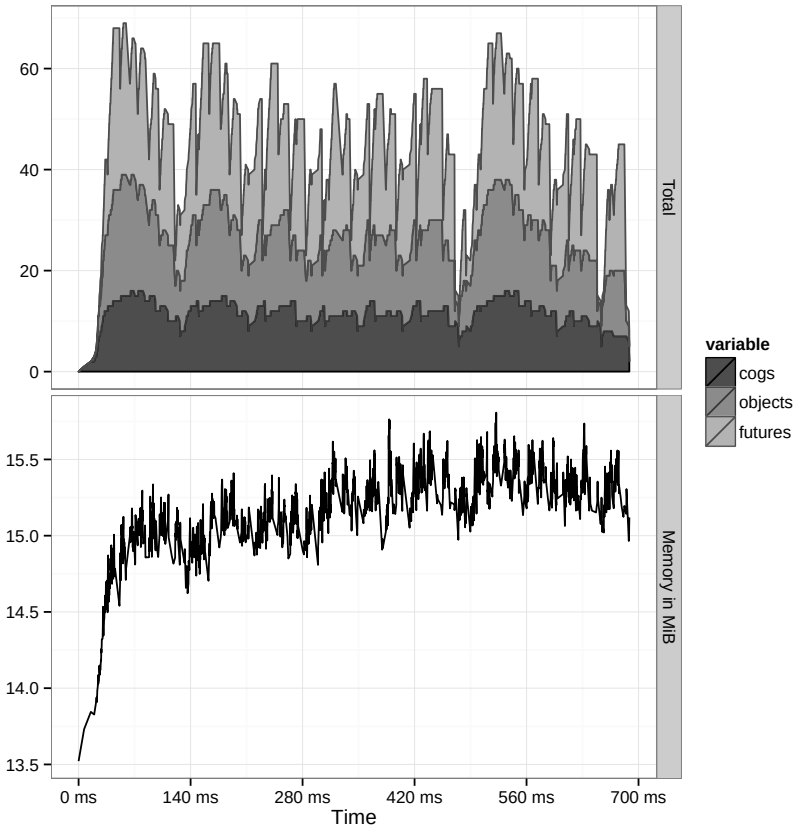

Table A.11: Ping Pong - Collect on count - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.2 MiB	15.2 MiB	15.8 MiB
COGs	0	12.1	12	16
Objects	0	14.9	15	24
Futures	0	9.4	10	23
Root futures	0	9.1	9	22
Processes	24	141.7	138	200
Φ	0.9 ‰	5.4 ‰	5.3 ‰	7.6 ‰

Table A.12: Ping Pong - Collect on count - Sweeps

	Min	Mean	Median	Max
Objects swept	0	4.7	6	10
Objects kept	3	12.4	12	22
Futures swept	2	14.2	15.5	22
Futures kept	0	0.8	1	1

Table A.13: Ping Pong - Collect on count - Intervals

	Mean	Median	Max	Total
Stop world	6958 µs	7587 µs	14215 µs	292253 µs
Mark	327 µs	323 µs	587 µs	13721 µs
Sweep	111 µs	119 µs	219 µs	4669 µs
Collection cycle	7396 µs	8040 µs	14572 µs	310643 µs
Collector only	438 µs	436 µs	635 µs	18390 µs
Mutator only	8904 µs	6508 µs	32655 µs	373969 µs

A.2.6 Trigger on Count or Timeout

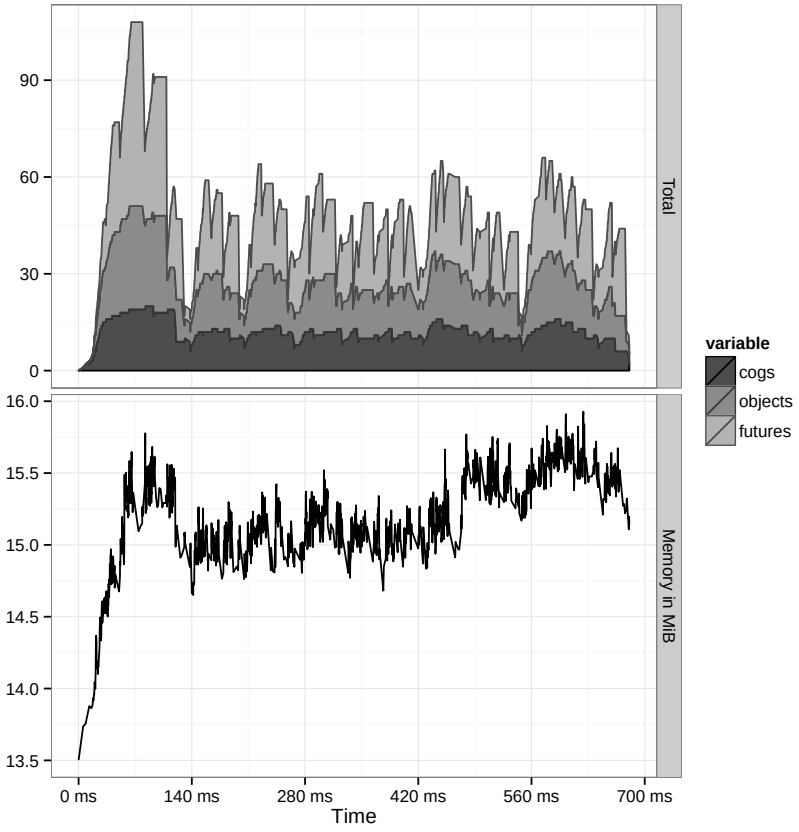

Table A.14: Ping Pong - Collect on count or time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.2 MiB	15.2 MiB	15.9 MiB
COGs	0	12.8	12	20
Objects	0	15.9	15	32
Futures	0	10.6	10	37
Root futures	0	9.4	8	27
Processes	24	145.8	145	207
Φ	0.9 ‰	5.6 ‰	5.5 ‰	7.9 ‰

Table A.15: Ping Pong - Collect on count or time - Sweeps

	Min	Mean	Median	Max
Objects swept	0	5.4	6	20
Objects kept	3	12.3	11	26
Futures swept	2	16.1	17	36
Futures kept	0	0.9	1	1

Table A.16: Ping Pong - Collect on count or time - Intervals

	Mean	Median	Max	Total
Stop world	7590 µs	7286 µs	16702 µs	280813 µs
Mark	349 µs	328 µs	682 µs	12920 µs
Sweep	124 µs	120 µs	354 µs	4573 µs
Collection cycle	8062 µs	7725 µs	17355 µs	298306 µs
Collector only	473 µs	440 µs	719 µs	17493 µs
Mutator only	10247 µs	8300 µs	29275 µs	379132 µs

A.2.7 Trigger on Count or Timeout with Stopping Running Tasks

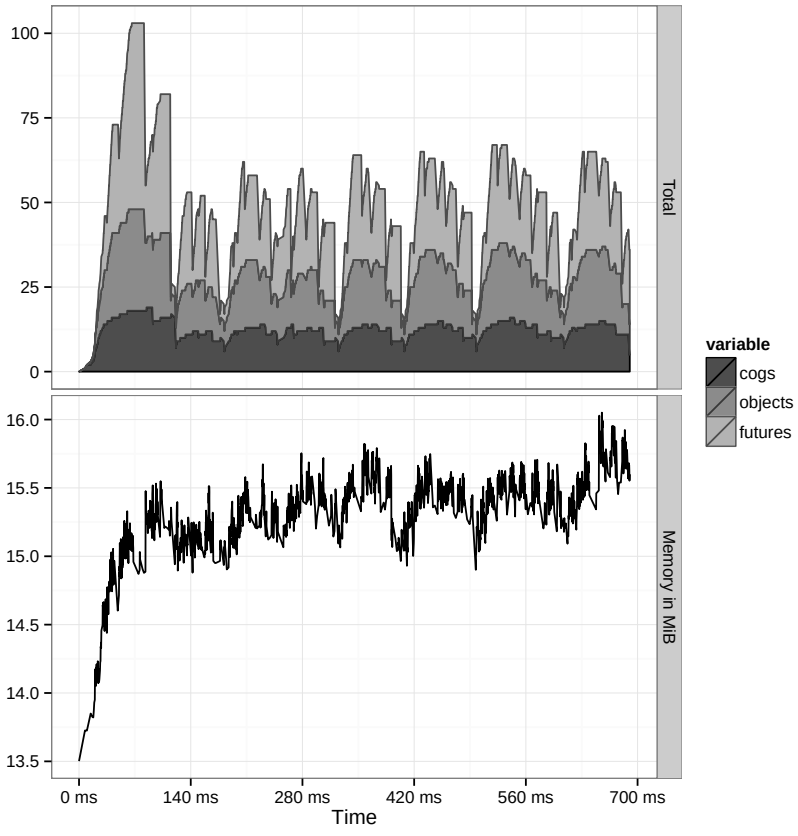

Table A.17: Ping Pong - Collect on count or time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.3 MiB	15.4 MiB	16.1 MiB
COGs	0	12.8	13	19
Objects	0	16.1	16	30
Futures	0	9.5	8	40
Root futures	0	10.3	10	25
Processes	24	148.1	148	212
Φ	0.9 ‰	5.6 ‰	5.6 ‰	8.1 ‰

Table A.18: Ping Pong - Collect on count or time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	0	4.8	4	20
Objects kept	4	13.3	14	25
Futures swept	2	14.4	14	40
Futures kept	0	0.2	0	1

Table A.19: Ping Pong - Collect on count or time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	8039 µs	8509 µs	17128 µs	321560 µs
Mark	357 µs	374 µs	650 µs	14286 µs
Sweep	115 µs	111 µs	361 µs	4613 µs
Collection cycle	8511 µs	8944 µs	17881 µs	340459 µs
Collector only	472 µs	462 µs	753 µs	18899 µs
Mutator only	8508 µs	6662 µs	31302 µs	340323 µs

A.3 Infinitely Looping Ping Pong

A.3.1 Never Triggering Collection

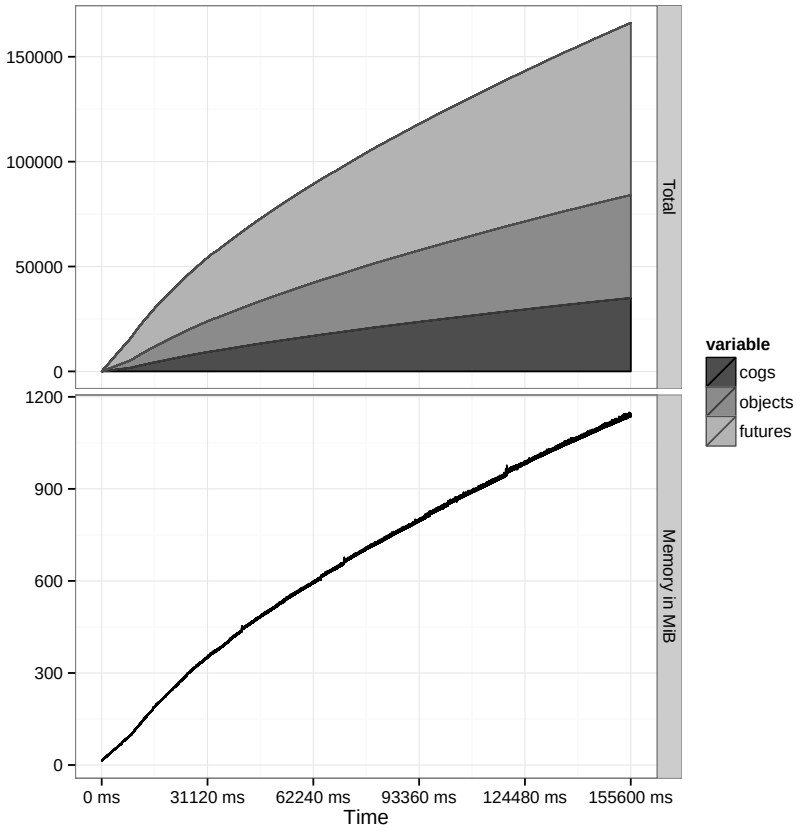

Table A.20: Infinite Ping Pong - Never Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	530.9 MiB	514.1 MiB	1149.3 MiB
COGs	0	15240	14349	35021
Objects	0	22430.2	21692.5	49051
Futures	0	25099.1	26530	42097
Root futures	0	16250.1	15017	39971
Processes	24	118586	113985	262143
Φ	0.9 ‰	4523.7 ‰	4348.2 ‰	10000 ‰

A.3.2 Triggering on Timeout

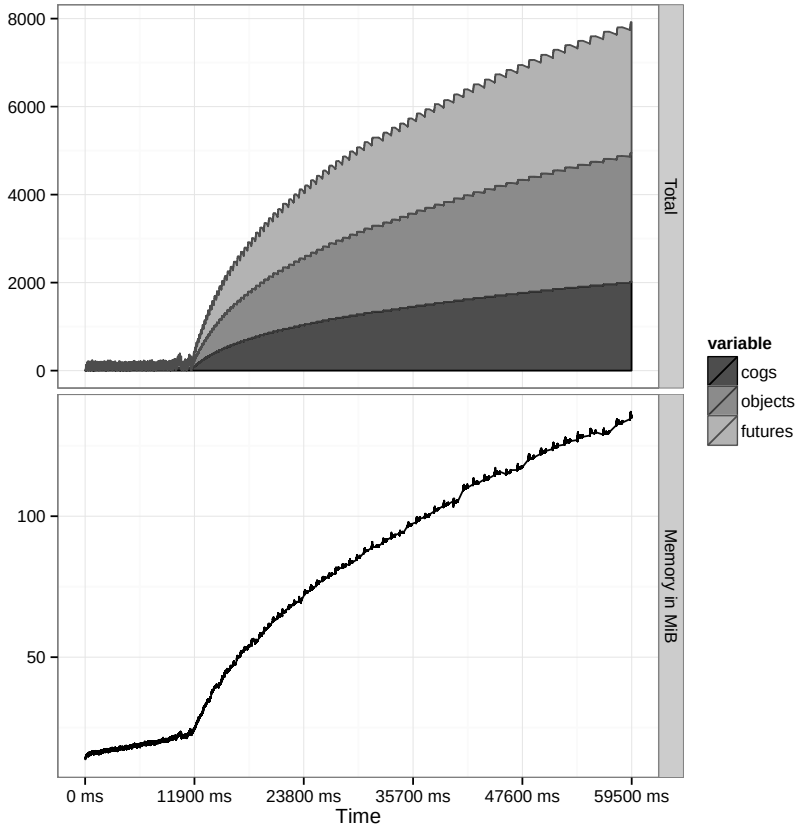

Table A.21: Infinite Ping Pong - Collect on time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	68.2 MiB	69.3 MiB	137.1 MiB
COGs	0	919.9	1006	2021
Objects	0	1342.5	1477	2929
Futures	0	54.2	55	184
Root futures	0	1327.2	1465	2919
Processes	24	8309.3	9234	16409
Φ	0.9 ‰	317 ‰	352.2 ‰	626 ‰

Table A.22: Infinite Ping Pong - Collect on time - Sweeps

	Min	Mean	Median	Max
Objects swept	6	27.9	28	80
Objects kept	3	609.1	28	2877
Futures swept	48	92.5	100	183
Futures kept	0	0.7	1	1

Table A.23: Infinite Ping Pong - Collect on time - Intervals

	Mean	Median	Max	Total
Stop world	107914 µs	12538 µs	678306 µs	18669067 µs
Mark	131812 µs	1737 µs	805210 µs	22803489 µs
Sweep	1128 µs	893 µs	4927 µs	195218 µs
Collection cycle	240854 µs	16132 µs	1362937 µs	41667774 µs
Collector only	132941 µs	2423 µs	807442 µs	22998707 µs
Mutator only	101906 µs	100964 µs	113549 µs	17629734 µs

Appendix B

Sequences Test Case

B.1 Source Code

```
module Sequences;
export *;

// Interface for generation of sequences
interface Sequence {
    Int next();
}

class NaturalNumbers implements Sequence {
    Int i = 0;

    Int next() {
        i = i + 1;
        return i;
    }
}

class Factorials implements Sequence {
    Int seq = 1;
    Int fact = 1;

    Int next() {
        fact = seq * fact;
        seq = seq + 1;
        return fact;
    }
}

class Fibonacci implements Sequence {
    Int prev = 0;
    Int prevPrev = 1;

    Int next() {
        Int n = prev + prevPrev;
        prevPrev = prev;
        prev = n;
        return n;
    }
}
```

```
class Squares implements Sequence {
    Int seq = 0;

    Int next() {
        seq = seq + 1;
        return seq * seq;
    }
}

class Cubes implements Sequence {
    Int seq = 0;

    Int next() {
        seq = seq + 1;
        return seq * seq * seq;
    }
}

{
    List<Sequence> sequences = Nil;
    Sequence s = new NaturalNumbers();
    sequences = Cons(s,sequences);
    s = new Fibonacci();
    sequences = Cons(s,sequences);
    s = new Factorials();
    sequences = Cons(s,sequences);
    s = new Squares();
    sequences = Cons(s,sequences);
    s = new Cubes();
    sequences = Cons(s,sequences);

    // Generate 1000th element of all sequences
    while (sequences != Nil) {
        Int i = 0;
        s = head(sequences);
        while (i < 1000) {
            s ! next();
            i = i + 1;
        }

        sequences = tail(sequences);
    }
}
```

B.2 Results

B.2.1 Never Triggering Collection

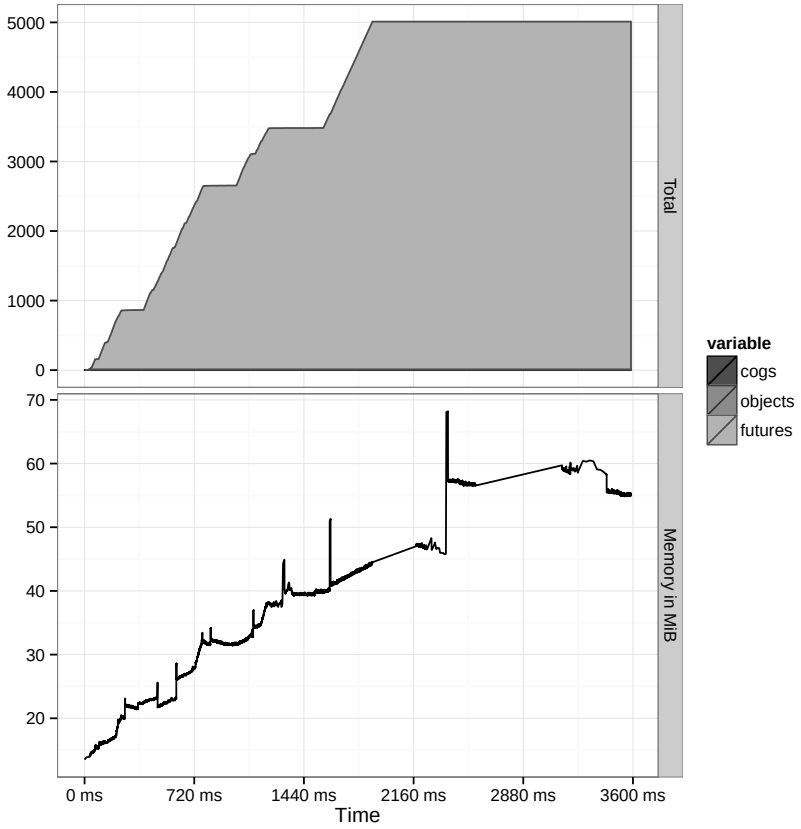

Table B.1: Sequences - Never Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	35.7 MiB	33.1 MiB	68.2 MiB
COGs	0	6	6	6
Objects	0	5	5	5
Futures	0	2029.5	1898.5	5000
Root futures	0	935.5	967	2000
Processes	24	3137.9	3371	6037
Φ	0.9 ‰	119.7 ‰	128.6 ‰	230.3 ‰

B.2.2 Always Triggering Collection

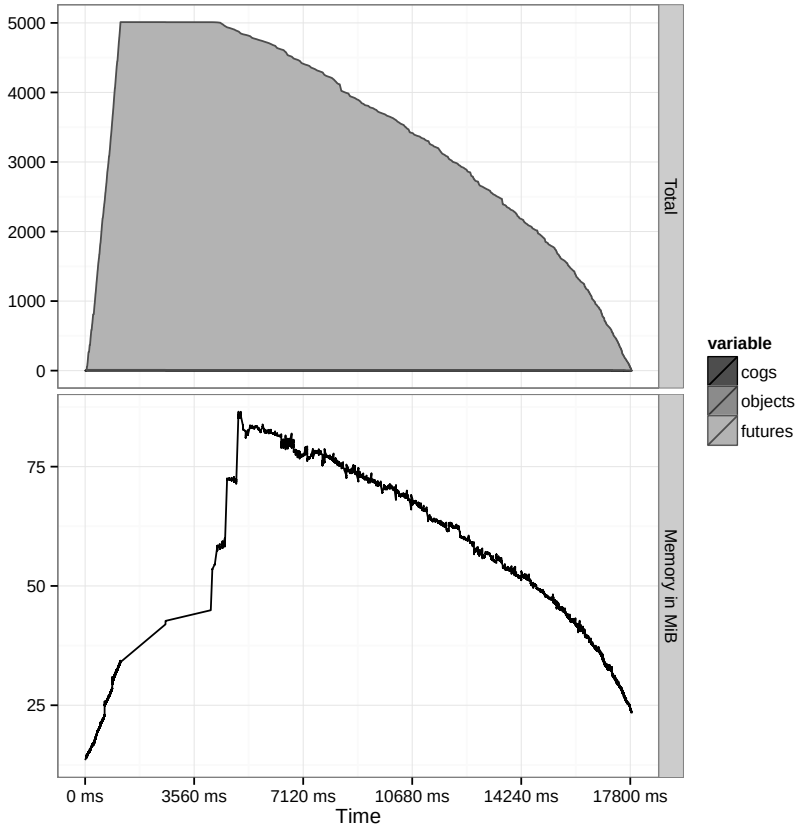

Table B.2: Sequences - Always Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	41.6 MiB	33.7 MiB	86.5 MiB
COGs	0	5.3	5	6
Objects	0	4.9	5	5
Futures	0	6	2	96
Root futures	0	2439.2	2411.5	5000
Processes	24	3890.3	3459	10040
Φ	0.9 ‰	148.4 ‰	132 ‰	383 ‰

Table B.3: Sequences - Always Collect - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	1
Objects kept	0	4.2	5	5
Futures swept	0	9.9	6	96
Futures kept	0	0	0	0

Table B.4: Sequences - Always Collect - Intervals

	Mean	Median	Max	Total
Stop world	4829 µs	1981 µs	1107051 µs	2443238 µs
Mark	29599 µs	18345 µs	1473534 µs	14977267 µs
Sweep	312 µs	300 µs	1244 µs	157869 µs
Collection cycle	34740 µs	21858 µs	2575248 µs	17578374 µs
Collector only	29911 µs	18692 µs	1474075 µs	15135136 µs
Mutator only	519 µs	239 µs	54419 µs	262574 µs

B.2.3 Trigger on Timeout

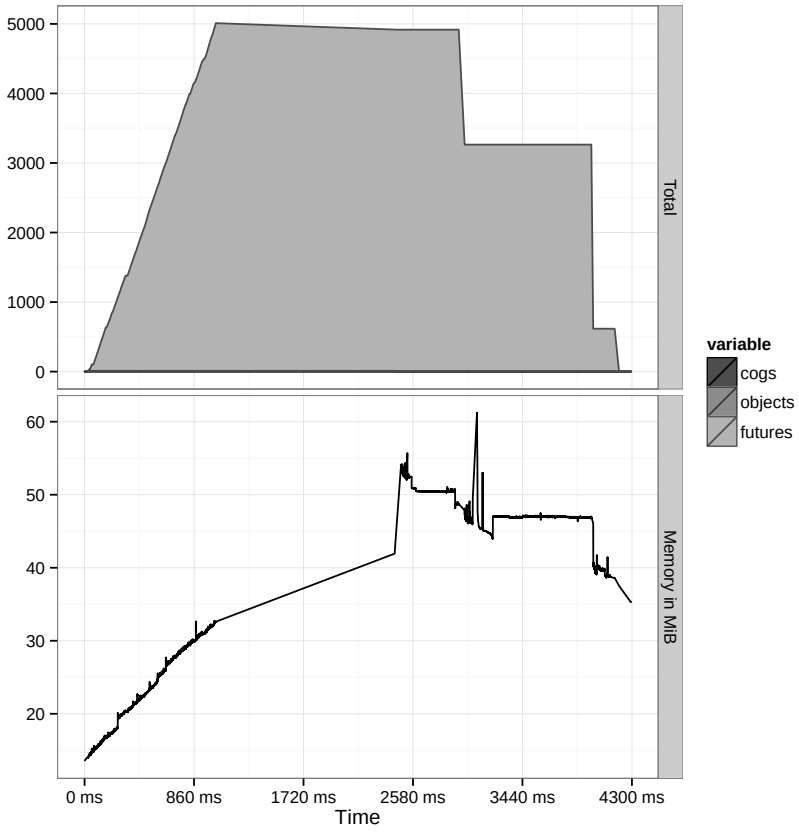

Table B.5: Sequences - Collect on time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	35.4 MiB	32.3 MiB	61.2 MiB
COGs	0	5.5	6	6
Objects	0	5	5	5
Futures	0	551.8	93	2647
Root futures	0	2402	2401	4907
Processes	24	3754.8	3902	9847
Φ	0.9 ‰	143.2 ‰	148.8 ‰	375.6 ‰

Table B.6: Sequences - Collect on time - Sweeps

	Min	Mean	Median	Max
Objects swept	0	1.2	0	5
Objects kept	0	3.8	5	5
Futures swept	93	1250	1130	2647
Futures kept	0	0	0	0

Table B.7: Sequences - Collect on time - Intervals

	Mean	Median	Max	Total
Stop world	582425 µs	648370 µs	931294 µs	2329700 µs
Mark	364064 µs	26022 µs	1404118 µs	1456258 µs
Sweep	2632 µs	2304 µs	5169 µs	10530 µs
Collection cycle	949122 µs	678765 µs	2336164 µs	3796488 µs
Collector only	366697 µs	30394 µs	1404870 µs	1466788 µs
Mutator only	100543 µs	100312 µs	101469 µs	402171 µs

B.2.4 Trigger on Timeout with Stopping Running Tasks

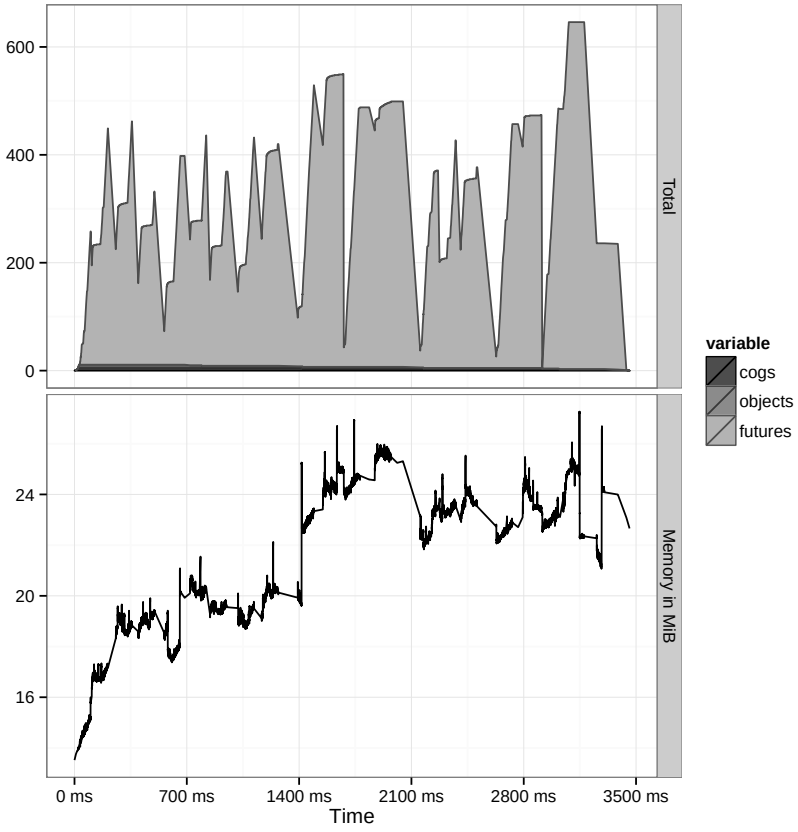

Table B.8: Sequences - Collect on time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	21.6 MiB	22.4 MiB	27.3 MiB
COGs	0	4.2	4	6
Objects	0	3.1	3	5
Futures	0	125.3	104	507
Root futures	0	193	166	643
Processes	24	446.6	393	1318
Φ	0.9 ‰	17 ‰	15 ‰	50.3 ‰

Table B.9: Sequences - Collect on time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.2	0	1
Objects kept	0	2.9	3	5
Futures swept	0	238.1	224	507
Futures kept	0	0	0	0

Table B.10: Sequences - Collect on time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	54606 µs	42809 µs	182148 µs	1146725 µs
Mark	7033 µs	5109 µs	22563 µs	147699 µs
Sweep	457 µs	415 µs	926 µs	9601 µs
Collection cycle	62096 µs	53257 µs	183463 µs	1304025 µs
Collector only	7490 µs	5849 µs	22775 µs	157300 µs
Mutator only	101705 µs	100169 µs	132022 µs	2135811 µs

B.2.5 Trigger on Count

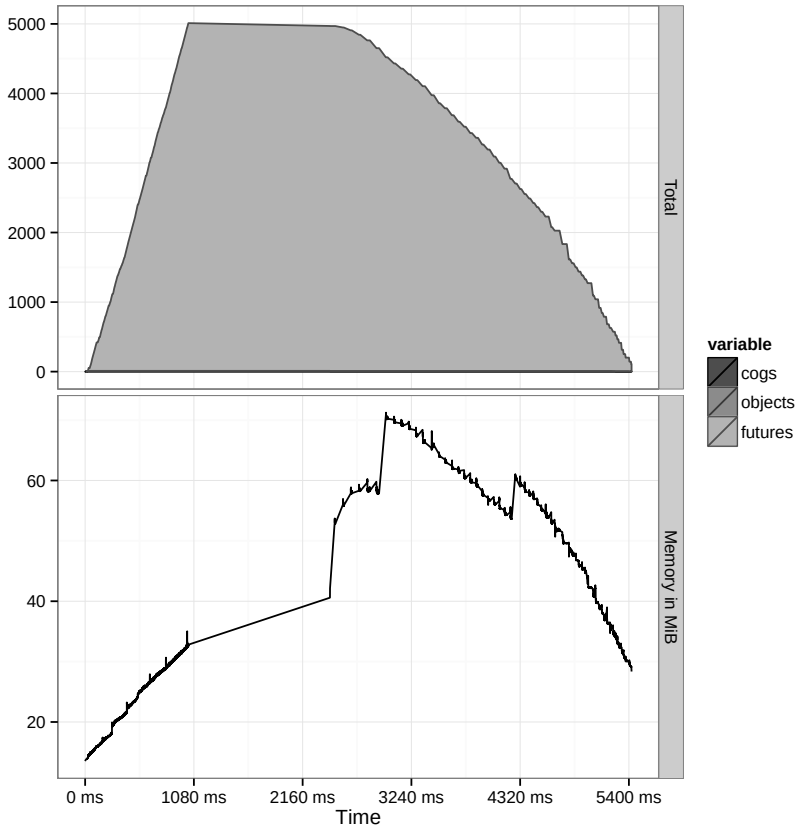

Table B.11: Sequences - Collect on count - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	38.5 MiB	32.1 MiB	71.2 MiB
COGs	0	5.4	5	6
Objects	0	4.9	5	5
Futures	0	45.8	41	221
Root futures	0	2478.7	2479	4959
Processes	24	3818.4	3468	9952
Φ	0.9 ‰	145.7 ‰	132.3 ‰	379.6 ‰

Table B.12: Sequences - Collect on count - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	2
Objects kept	2	4.8	5	5
Futures swept	21	81.6	71.5	221
Futures kept	0	0	0	0

Table B.13: Sequences - Collect on count - Intervals

	Mean	Median	Max	Total
Stop world	31129 µs	12796 µs	987860 µs	1867761 µs
Mark	54358 µs	29976 µs	1403626 µs	3261471 µs
Sweep	592 µs	646 µs	1196 µs	35545 µs
Collection cycle	86080 µs	46778 µs	2392099 µs	5164777 µs
Collector only	54950 µs	30538 µs	1404239 µs	3297016 µs
Mutator only	4310 µs	2492 µs	50948 µs	258629 µs

B.2.6 Trigger on Count or Timeout

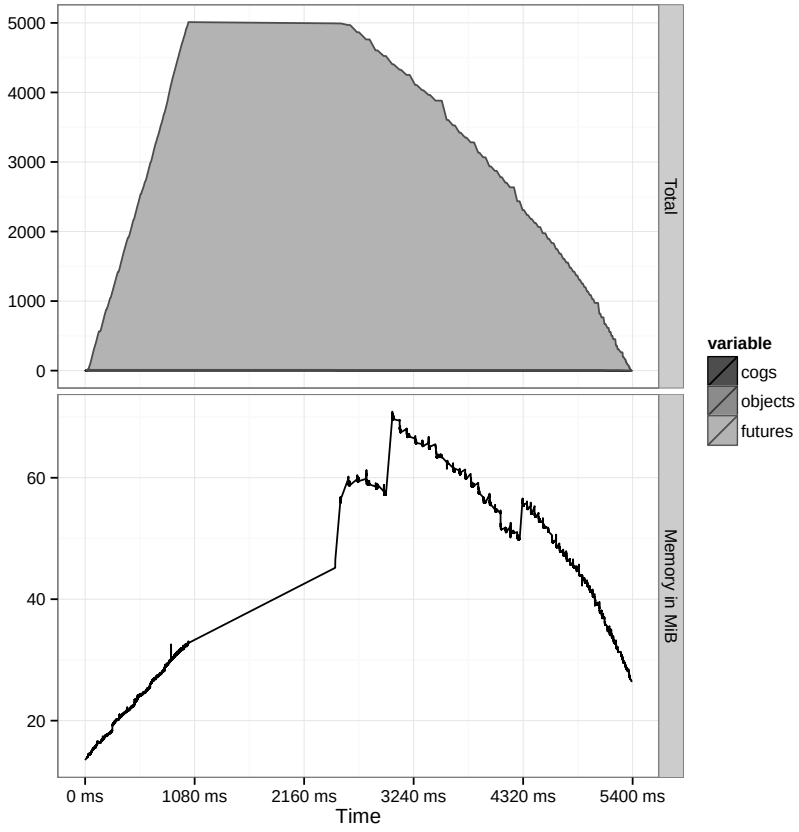

Table B.14: Sequences - Collect on count or time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	37.9 MiB	32 MiB	70.9 MiB
COGs	0	5.4	5	6
Objects	0	4.9	5	5
Futures	0	35.1	18	277
Root futures	0	2471	2464	4982
Processes	24	3793.4	3373	9990
Φ	0.9 ‰	144.7 ‰	128.7 ‰	381.1 ‰

Table B.15: Sequences - Collect on count or time - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.1	0	1
Objects kept	0	4.4	5	5
Futures swept	16	74.6	66	277
Futures kept	0	0	0	0

Table B.16: Sequences - Collect on count or time - Intervals

	Mean	Median	Max	Total
Stop world	27904 µs	11374 µs	984836 µs	1869601 µs
Mark	47796 µs	23160 µs	1446317 µs	3202303 µs
Sweep	512 µs	464 µs	1201 µs	34271 µs
Collection cycle	76212 µs	35753 µs	2431724 µs	5106175 µs
Collector only	48307 µs	23622 µs	1446888 µs	3236574 µs
Mutator only	4198 µs	2646 µs	53475 µs	281262 µs

B.2.7 Trigger on Count or Timeout with Stopping Running Tasks

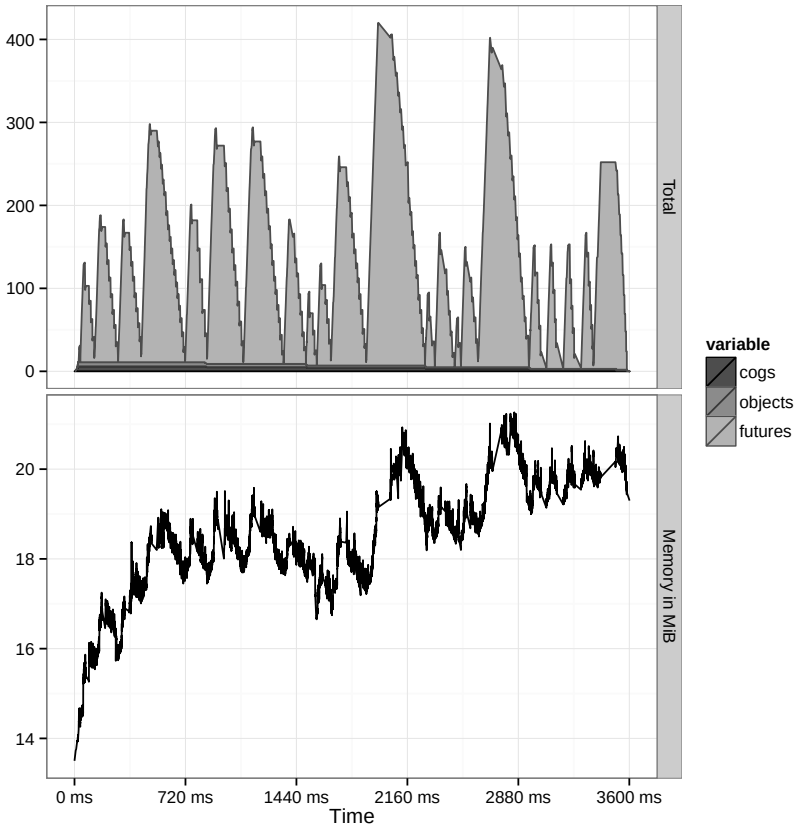

Table B.17: Sequences - Collect on count or time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	18.5 MiB	18.5 MiB	21.3 MiB
COGs	0	4.1	4	6
Objects	0	3.1	3	5
Futures	0	10.5	8	49
Root futures	0	114	96	402
Processes	24	246.5	197	848
Φ	0.9 ‰	9.4 ‰	7.5 ‰	32.3 ‰

Table B.18: Sequences - Collect on count or time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	1
Objects kept	0	3.2	3	5
Futures swept	9	26.2	25	49
Futures kept	0	0	0	0

Table B.19: Sequences - Collect on count or time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	8264 µs	5673 µs	69794 µs	1578457 µs
Mark	2304 µs	1687 µs	10172 µs	440101 µs
Sweep	87 µs	84 µs	193 µs	16651 µs
Collection cycle	10656 µs	8035 µs	80100 µs	2035209 µs
Collector only	2391 µs	1774 µs	10306 µs	456752 µs
Mutator only	8109 µs	3260 µs	143453 µs	1548880 µs

Appendix C

Prime Sieve Test Case

C.1 Source Code

Listing C.1: Full source code for Prime Sieve Test Case

```
module PrimeSieve;
export *;

// Tests primality by division with list of earlier primes
def Bool is_prime(Int n, List<Int> earlierPrimes) =
  case earlierPrimes {
    Nil => True;
    Cons(p, rest) => if (n % p == 0)
                     then False
                     else is_prime(n, rest);
  };

interface PrimeSieve {
  Int getPrime();
}

// Sieve that generates the nth prime
class PrimeSieve(Int n) implements PrimeSieve {
  Int nthPrime = 0;

  {
    List<Int> primes = Cons(2, Nil);
    Int i = 1;
    Int x = 3;
    while (i < n) {
      if (is_prime(x, primes)) {
        primes = Cons(x, primes);
        i = i + 1;
      }
      x = x + 2;
    }
    nthPrime = head(primes);
  }

  Int getPrime() {
    return nthPrime;
  }
```

```
}

// Main block
// Creates 16 sieves, then waits of them to finish
{
  List<Fut<Int>> primes = Nil;
  Int i = 1;
  while (i <= 16) {
    PrimeSieve sieve = new PrimeSieve(i * 64);
    Fut<Int> result = sieve!getPrime();
    primes = Cons(result, primes);
    i = i + 1;
  }

  while (primes != Nil) {
    Fut<Int> next = head(primes);
    await next?;
    primes = tail(primes);
  }
}
```

C.2 Results

C.2.1 Never Triggering Collection

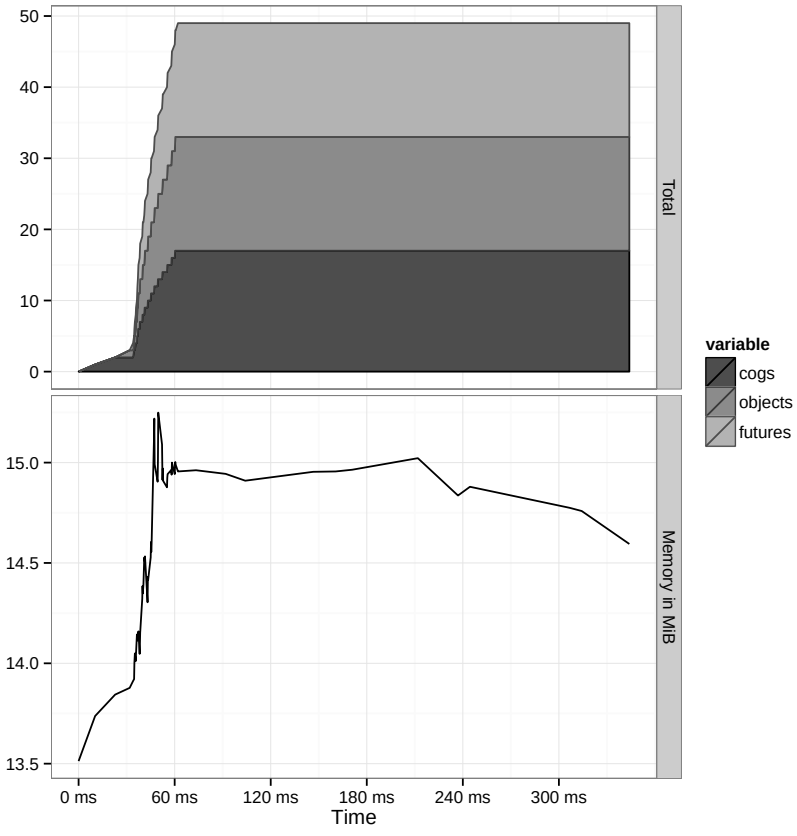

Table C.1: Prime Sieve - Never Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.6 MiB	14.6 MiB	15.2 MiB
COGs	0	10.8	11	17
Objects	0	9.5	10	16
Futures	0	2.9	2	16
Root futures	0	6.3	6.5	12
Processes	24	78.1	82.5	114
Φ	0.9 ‰	3 ‰	3.1 ‰	4.3 ‰

C.2.2 Always Triggering Collection

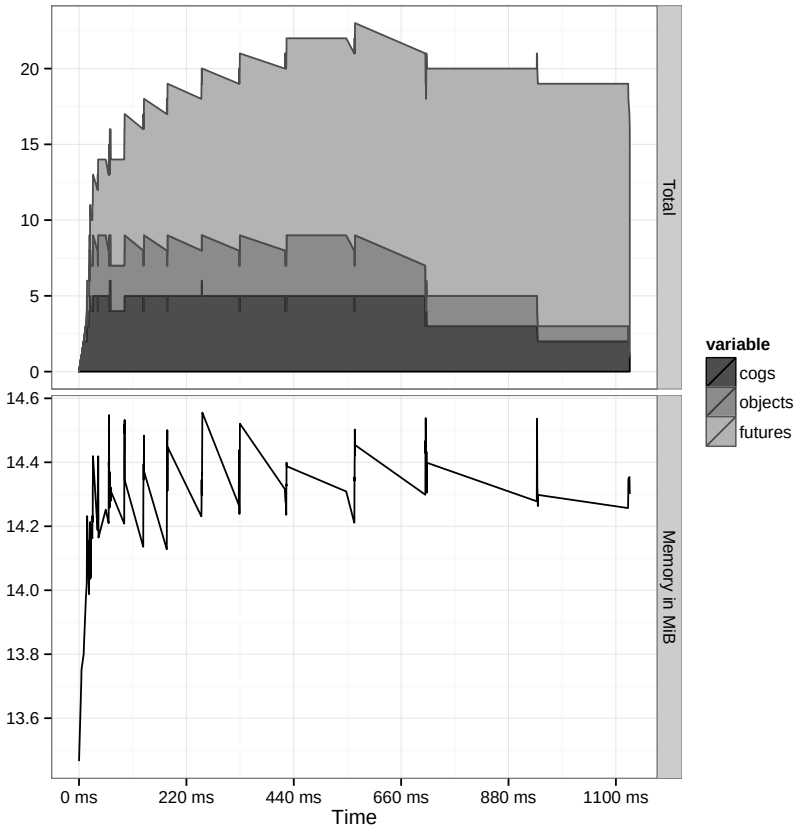

Table C.2: Prime Sieve - Always Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.3 MiB	14.3 MiB	14.6 MiB
COGs	0	4.2	5	6
Objects	0	2.7	3	4
Futures	0	7.2	7	16
Root futures	0	1.5	2	3
Processes	24	54.8	58	67
Φ	0.9 ‰	2.1 ‰	2.2 ‰	2.6 ‰

Table C.3: Prime Sieve - Always Collect - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.7	1	2
Objects kept	0	2	2	3
Futures swept	0	0	0	1
Futures kept	0	7.7	7.5	16

Table C.4: Prime Sieve - Always Collect - Intervals

	Mean	Median	Max	Total
Stop world	50272 µs	16308 µs	226339 µs	1105979 µs
Mark	167 µs	160 µs	360 µs	3677 µs
Sweep	38 µs	34 µs	98 µs	830 µs
Collection cycle	50477 µs	16464 µs	226501 µs	1110486 µs
Collector only	205 µs	192 µs	367 µs	4507 µs
Mutator only	817 µs	248 µs	4522 µs	17975 µs

C.2.3 Trigger on Timeout

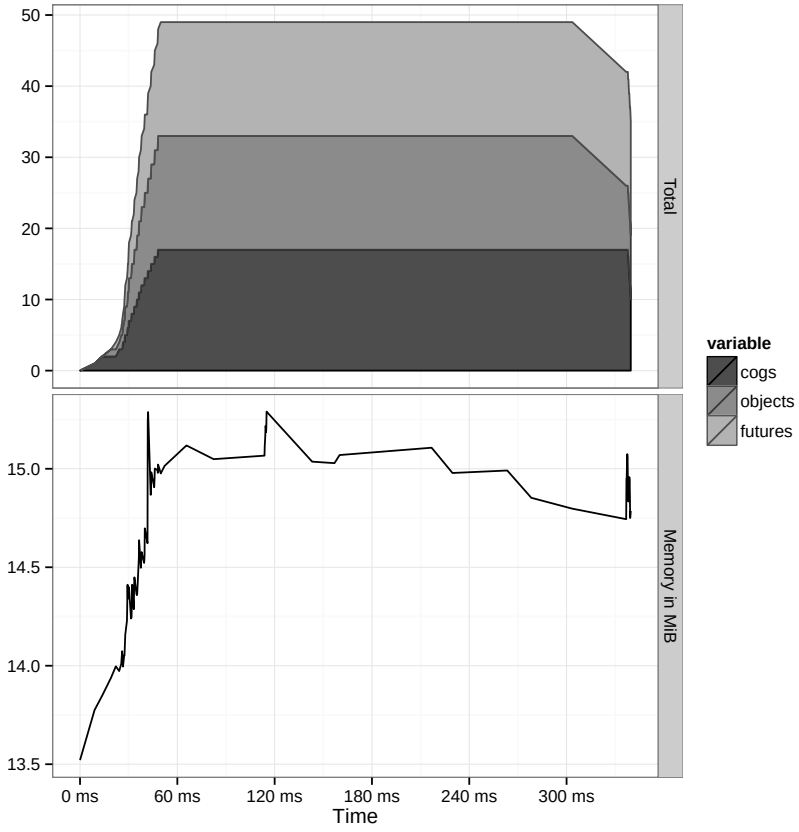

Table C.5: Prime Sieve - Collect on time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.7 MiB	14.8 MiB	15.3 MiB
COGs	0	12	13	17
Objects	0	9.9	9	16
Futures	0	4.8	3	16
Root futures	0	6.2	7	13
Processes	24	80.8	78	116
Φ	0.9 ‰	3.1 ‰	3 ‰	4.4 ‰

Table C.6: Prime Sieve - Collect on time - Sweeps

	Min	Mean	Median	Max
Objects swept	7	7	7	7
Objects kept	9	9	9	9
Futures swept	0	0	0	0
Futures kept	7	7	7	7

Table C.7: Prime Sieve - Collect on time - Intervals

	Mean	Median	Max	Total
Stop world	222588 µs	222588 µs	222588 µs	222588 µs
Mark	272 µs	272 µs	272 µs	272 µs
Sweep	151 µs	151 µs	151 µs	151 µs
Collection cycle	223011 µs	223011 µs	223011 µs	223011 µs
Collector only	423 µs	423 µs	423 µs	423 µs
Mutator only	113730 µs	113730 µs	113730 µs	113730 µs

C.2.4 Trigger on Timeout with Stopping Running Tasks

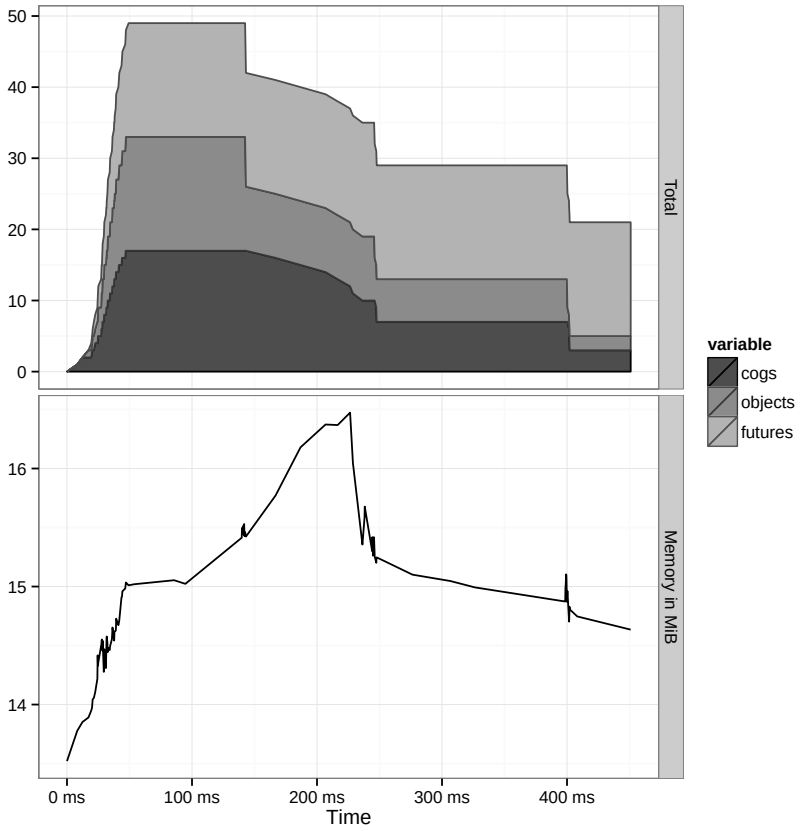

Table C.8: Prime Sieve - Collect on time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.9 MiB	15 MiB	16.5 MiB
COGs	0	10.4	10	17
Objects	0	8.9	9	16
Futures	0	5.8	7	16
Root futures	0	6.2	6	13
Processes	24	82.5	88	116
Φ	0.9 ‰	3.1 ‰	3.4 ‰	4.4 ‰

Table C.9: Prime Sieve - Collect on time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	3	4.7	4	7
Objects kept	2	5.7	6	9
Futures swept	0	0	0	0
Futures kept	7	10.3	10	14

Table C.10: Prime Sieve - Collect on time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	2055 µs	1773 µs	2881 µs	6165 µs
Mark	235 µs	197 µs	344 µs	704 µs
Sweep	100 µs	88 µs	148 µs	301 µs
Collection cycle	2390 µs	2035 µs	3373 µs	7170 µs
Collector only	335 µs	262 µs	492 µs	1005 µs
Mutator only	131047 µs	139763 µs	152415 µs	393142 µs

C.2.5 Trigger on Count

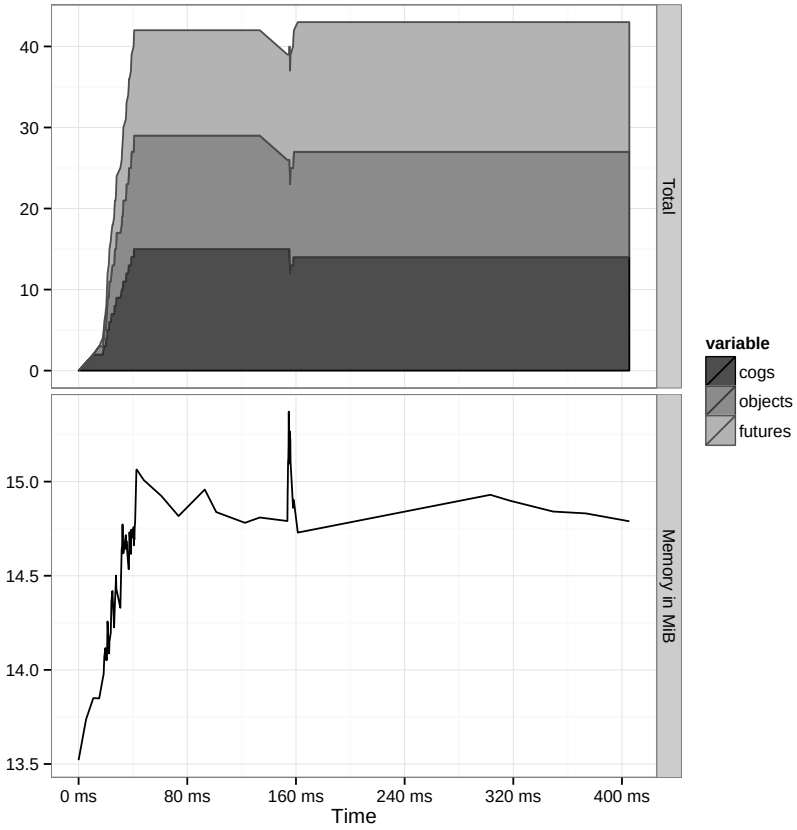

Table C.11: Prime Sieve - Collect on count - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.6 MiB	14.7 MiB	15.4 MiB
COGs	0	11	13	15
Objects	0	9.4	11	14
Futures	0	4.1	3	16
Root futures	0	5.7	5	10
Processes	24	77	84.5	101
Φ	0.9 ‰	2.9 ‰	3.2 ‰	3.9 ‰

Table C.12: Prime Sieve - Collect on count - Sweeps

	Min	Mean	Median	Max
Objects swept	3	3	3	3
Objects kept	11	11	11	11
Futures swept	0	0	0	0
Futures kept	3	3	3	3

Table C.13: Prime Sieve - Collect on count - Intervals

	Mean	Median	Max	Total
Stop world	112374 µs	112374 µs	112374 µs	112374 µs
Mark	350 µs	350 µs	350 µs	350 µs
Sweep	78 µs	78 µs	78 µs	78 µs
Collection cycle	112802 µs	112802 µs	112802 µs	112802 µs
Collector only	428 µs	428 µs	428 µs	428 µs
Mutator only	40875 µs	40875 µs	40875 µs	40875 µs

C.2.6 Trigger on Count or Timeout

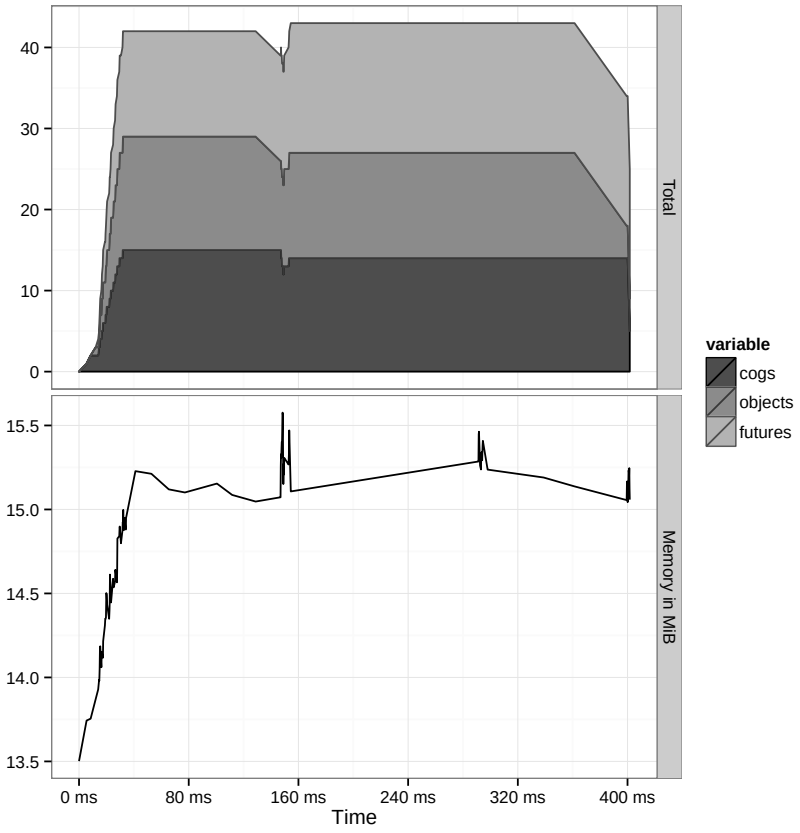

Table C.14: Prime Sieve - Collect on count or time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	14.9 MiB	15 MiB	15.6 MiB
COGs	0	11.1	13	15
Objects	0	9	11	14
Futures	0	5.9	3	16
Root futures	0	5.3	4	10
Processes	24	77.9	86	101
Φ	0.9 ‰	3 ‰	3.3 ‰	3.9 ‰

Table C.15: Prime Sieve - Collect on count or time - Sweeps

	Min	Mean	Median	Max
Objects swept	3	6	6	9
Objects kept	4	7.5	7.5	11
Futures swept	0	0	0	0
Futures kept	3	7.5	7.5	12

Table C.16: Prime Sieve - Collect on count or time - Intervals

	Mean	Median	Max	Total
Stop world	111046 µs	111046 µs	114434 µs	222091 µs
Mark	298 µs	298 µs	360 µs	595 µs
Sweep	132 µs	132 µs	177 µs	265 µs
Collection cycle	111476 µs	111476 µs	114882 µs	222951 µs
Collector only	430 µs	430 µs	448 µs	860 µs
Mutator only	88203 µs	88203 µs	144370 µs	176406 µs

C.2.7 Trigger on Count or Timeout with Stopping Running Tasks

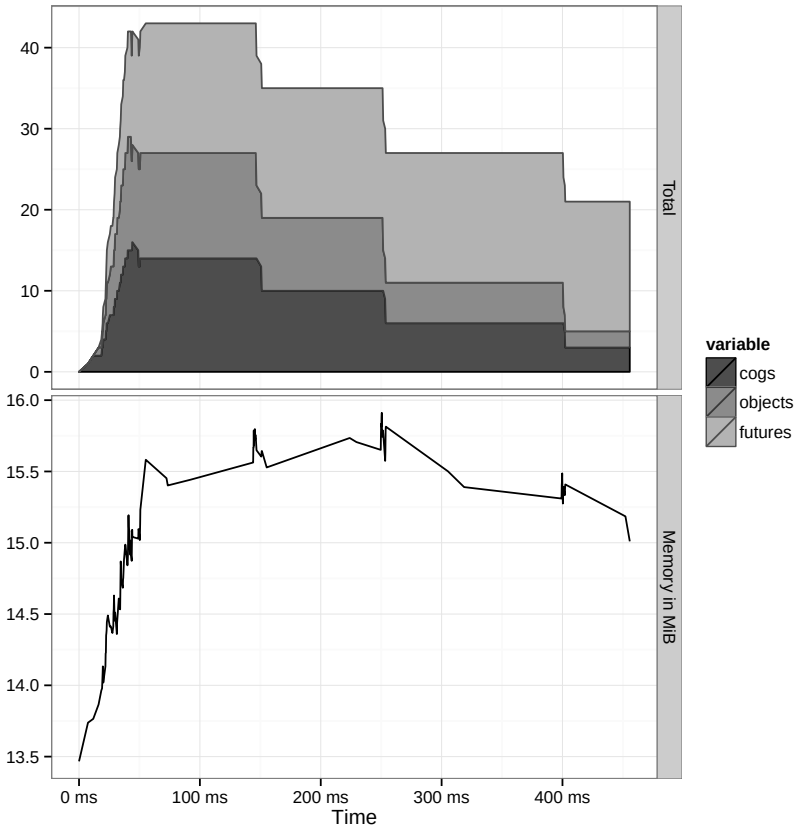

Table C.17: Prime Sieve - Collect on count or time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	15.1 MiB	15.1 MiB	15.9 MiB
COGs	0	10.2	10	16
Objects	0	8.8	9	14
Futures	0	5.4	3	16
Root futures	0	6.6	7	13
Processes	24	83.3	88	110
Φ	0.9 ‰	3.2 ‰	3.4 ‰	4.2 ‰

Table C.18: Prime Sieve - Collect on count or time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	3	3.5	3.5	4
Objects kept	2	6.8	7	11
Futures swept	0	0	0	0
Futures kept	3	8.8	9	14

Table C.19: Prime Sieve - Collect on count or time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	2019 µs	2038 µs	2678 µs	8077 µs
Mark	267 µs	266 µs	377 µs	1069 µs
Sweep	92 µs	94 µs	115 µs	370 µs
Collection cycle	2379 µs	2410 µs	3123 µs	9516 µs
Collector only	360 µs	371 µs	445 µs	1439 µs
Mutator only	97820 µs	101641 µs	147563 µs	391281 µs

Appendix D

Indexing Test Case

D.1 Source Code

Listing D.1: Indexing Test Case

```
module Indexing;
import * from ABS.DC;
import * from ABS.Meta;

// * Prefix indexing

// A model of a MapReduce job.  The input is a list of docments
// represented as a pair of (filename, contents),
// where contents is a wordlist.  The output is a list of
// (prefix, filenames), giving the occurrences of all prefixes
// of all words.

// The code follows the general pattern of MapReduce
// and is decomposed for easy modification via deltas.
// Product lines and deltas are not included in this copy.

type InKeyType = String;          // filename
type InValueType = String;        // file contents
type OutKeyType = String;         // prefix
type OutValueType = String;       // filename

def Int max_prefix_length() = 12;

def String listToString(List<String> l) =
  case l {
    Nil => "";
    Cons(x, rest) => x + ", " + listToString(rest);
  };
def String resultToString(Pair<String, List<String>> p) =
  case p {
    Pair(x,y) => x + ": " + listToString(y);
  };

interface Worker {
  // invoked by MapReduce component
  List<Pair<OutKeyType, OutValueType>>
    invokeMap(InKeyType key, InValueType value);
```

```
  // invoked by MapReduce component
  List<OutValueType> invokeReduce(OutKeyType key,
                                  List<OutValueType> value);
}

interface MapReduce {
  // invoked by client
  List<Pair<OutKeyType, List<OutValueType>>>
    mapReduce(List<Pair<InKeyType, InValueType>> documents);
  // invoked by workers
  Unit finished(Worker w);
}

class Worker(MapReduce master) implements Worker {
  List<Pair<OutKeyType, OutValueType>> mapResults = Nil;
  List<OutValueType> reduceResults = Nil;

  // The methods in this section can be overridden via deltas.
  // Map and reduce should not change the state of the object
  // and should not contain any cost annotations.
  Unit map(InKeyType key, InValueType value) {
    String content = value;
    Int begin = 0;
    Int max = strlen(content);
    Int end = begin;
    while (begin < max) {
      while (substr(content, begin, 1) == " " && begin < max) {
        begin = begin + 1;
      }
      // got a word boundary
      end = begin + 1;
      while (end - begin < max_prefix_length() && end <= max) {
        this.emitMapResult(substr(content, begin, end - begin),
                           key);
        end = end + 1;
      }
      while (substr(content, begin, 1) != " " && begin < max - 1) {
        begin = begin + 1;
      }
      begin = begin + 1;
    }
  }

  Unit reduce(OutKeyType key, List<OutValueType> value) {
    // Remove duplicates in occurrence list: convert into set.
    Set<OutValueType> resultset = set(value);
    while (~(resultset == EmptySet)) {
      OutValueType file = take(resultset);
      resultset = remove(resultset, file);
      this.emitReduceResult(file);
    }
  }

  // These methods can be overridden in deltas to contain cost
  // annotations relating to the respective phases of the map or
  // reduce step.  Any side effects should only be on state
  //  introduced same delta that replaced the default method.
  Unit onMapStart(InKeyType key, InValueType value) {
    skip;
```

```
  }
  Unit onMapEmit(OutKeyType key, OutValueType value) {
    skip;
  }
  Unit onMapFinish() {
    skip;
  }
  Unit onReduceStart(OutKeyType key, List<OutValueType> value) {
    skip;
  }
  Unit onReduceEmit(OutValueType value) {
    skip;
  }
  Unit onReduceFinish() {
    skip;
  }

  List<Pair<OutKeyType, OutValueType>> invokeMap(InKeyType key,
        InValueType value) {
    mapResults = Nil;
    this.onMapStart(key, value);
    this.map(key, value);
    this.onMapFinish();
    master!finished(this);
    List<Pair<OutKeyType, OutValueType>> result = mapResults;
    mapResults = Nil;
    return result;
  }

  List<OutValueType> invokeReduce(OutKeyType key,
        List<OutValueType> value) {
    reduceResults = Nil;
    this.onReduceStart(key, value);
    this.reduce(key, value);
    this.onReduceFinish();
    master!finished(this);
    List<OutValueType> result = reduceResults;
    reduceResults = Nil;
    return result;
  }

  Unit emitMapResult(OutKeyType key, OutValueType value) {
    this.onMapEmit(key, value);
    mapResults = Cons(Pair(key, value), mapResults);
  }
  Unit emitReduceResult(OutValueType value) {
    this.onReduceEmit(value);
    reduceResults = Cons(value, reduceResults);
  }
}

// This class contains the MapReduce machinery.
// Any deployment decisions (number of machines, etc.)
// can be customized via deltas.
class MapReduce implements MapReduce {
  Set<Worker> workers = EmptySet;
  Int nWorkers = 0;

  // This method obtains a Worker object.  Any VM creation, load
```

```
// balancing, accounting etc. goes on here.  Any side effects
// should only modify state that is introduced in the same delta.
Worker getWorker() {
  Worker w = null;
  if (emptySet(workers)) {
    w = new Worker(this);
    nWorkers = nWorkers + 1;
    } else {
    w = take(workers);
    workers = remove(workers, w);
  }
  return w;
}

// This method registers a worker as idle.
// It is called by the worker itself.
// Any side effects should only modify state that is
// introduced in the same delta.
Unit finished(Worker w) {
  workers = insertElement(workers, w);
}

List<Pair<OutKeyType, List<OutValueType>>>
      mapReduce(List<Pair<InKeyType, InValueType>> items) {
  Set<Fut<List<Pair<OutKeyType, OutValueType>>>> fMapResults
      = EmptySet;
  ABS.StdLib.Map<OutKeyType, List<OutValueType>> intermediates
      = EmptyMap;
  Set<Pair<OutKeyType, Fut<List<OutValueType>>>> fReduceResults
      = EmptySet;
  List<Pair<OutKeyType, List<OutValueType>>> result = Nil;

  while (~isEmpty(items)) {
    Pair<InKeyType, InValueType> item = head(items);
    items = tail(items);
    Worker w = this.getWorker();
    String key = fst(item);
    InValueType value = snd(item);
    // "Map, written by the user, takes an input pair and produces a
    // set of intermediate key/value pairs." [MapReduce, pg. 2]
    Fut<List<Pair<OutKeyType, OutValueType>>> fMap
        = w!invokeMap(key, value);
    fMapResults = insertElement(fMapResults, fMap);
  }
  while (~emptySet(fMapResults)) {
    // "The MapReduce library groups together all intermediate values
    // associated with the same intermediate key I ..." [ditto]
    Fut<List<Pair<OutKeyType, OutValueType>>> fMapResult
        = take(fMapResults);
    fMapResults = remove(fMapResults, fMapResult);
    await fMapResult?;
    List<Pair<OutKeyType, OutValueType>> mapResult
        = fMapResult.get;
    while (~isEmpty(mapResult)) {
      Pair<OutKeyType, OutValueType> keyValuePair
          = head(mapResult);
      mapResult = tail(mapResult);
      List<OutValueType> inter
          = lookupDefault(intermediates, fst(keyValuePair), Nil);
```

```
        intermediates = put(intermediates, fst(keyValuePair),
          Cons(snd(keyValuePair), inter));
      }
    }
    // "... and passes them to the Reduce function.  The Reduce
    // function, also written by the user, accepts an intermediate key I
    // and a set of values for that key. It merges together these values
    // to form a possibly smaller set of values.  Typically just zero or
    // one outpout value is produced per Reduce invocation." [ditto]
    Set<OutKeyType> keys = keys(intermediates);
    while(~emptySet(keys)) {
      OutKeyType key = take(keys);
      keys = remove(keys, key);
      List<OutValueType> values = lookupUnsafe(intermediates, key);
      Worker w = this.getWorker();
      Unit x = println("Reduce");
      Fut<List<OutValueType>> fReduce = w!invokeReduce(key, values);
      fReduceResults
          = insertElement(fReduceResults, Pair(key, fReduce));
    }
    while (~emptySet(fReduceResults)) {
      Pair<OutKeyType, Fut<List<OutValueType>>> reduceResult
          = take(fReduceResults);
      fReduceResults = remove(fReduceResults, reduceResult);
      OutKeyType key = fst(reduceResult);
      Fut<List<OutValueType>> fValues = snd(reduceResult);
      await fValues?;
      List<OutValueType> values = fValues.get;
      result = Cons(Pair(key, values), result);
    }
    return result;
  }

}

class Client(MapReduce m) {
  List<Pair<InKeyType, InValueType>> inputs =
    list[Pair("paul_clifford.txt",
          "it was a dark and stormy night"),
      Pair("tale_of_two_cities.txt",
        "it was the best of times it was the worst of times"),
      Pair("neuromancer.txt",
        "the sky above the port was the color of television tuned to a dead channel")
      ];
  List<Pair<OutKeyType, List<OutValueType>>> result = Nil;
  Unit run() {
    result = await m!mapReduce(inputs);
    List<Pair<OutKeyType, List<OutValueType>>> it = result;
    while (it != Nil) {
      Unit x = println(resultToString(head(it)));
      it = tail(it);
    }
  }
}

{
  MapReduce m = new MapReduce();
  new local Client(m);
}
```

D.2 Results

D.2.1 Never Triggering Collection

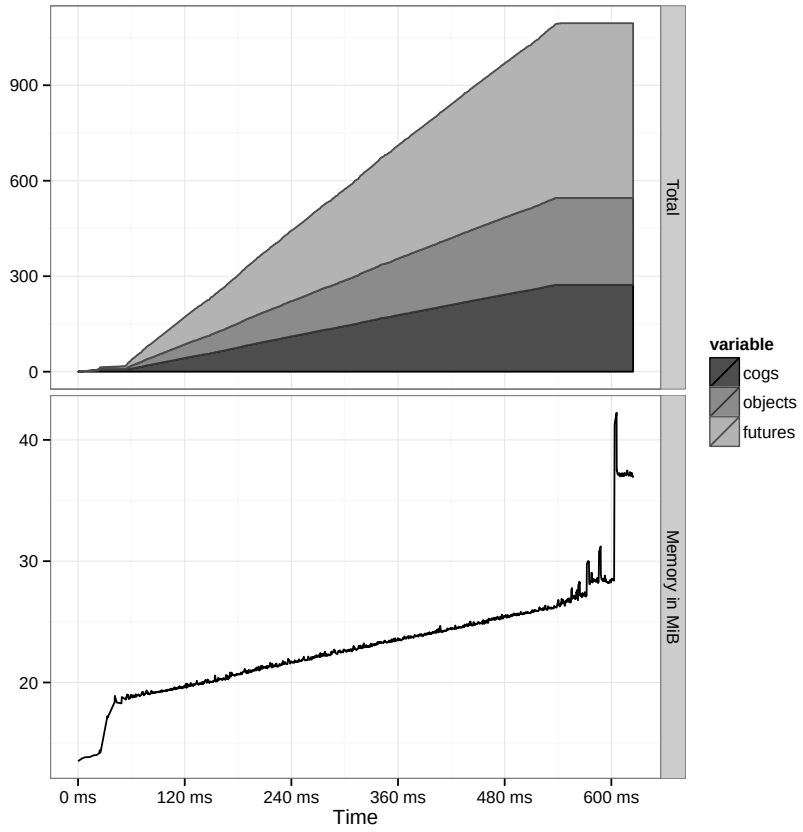

Table D.1: Indexing - Never Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	23.5 MiB	23.2 MiB	42.2 MiB
COGs	0	159.5	164	273
Objects	0	159.3	164	273
Futures	0	183.8	166	549
Root futures	0	135.5	135	272
Processes	24	959.4	1010	1665
Φ	0.9 ‰	36.6 ‰	38.5 ‰	63.5 ‰

D.2.2 Always Triggering Collection

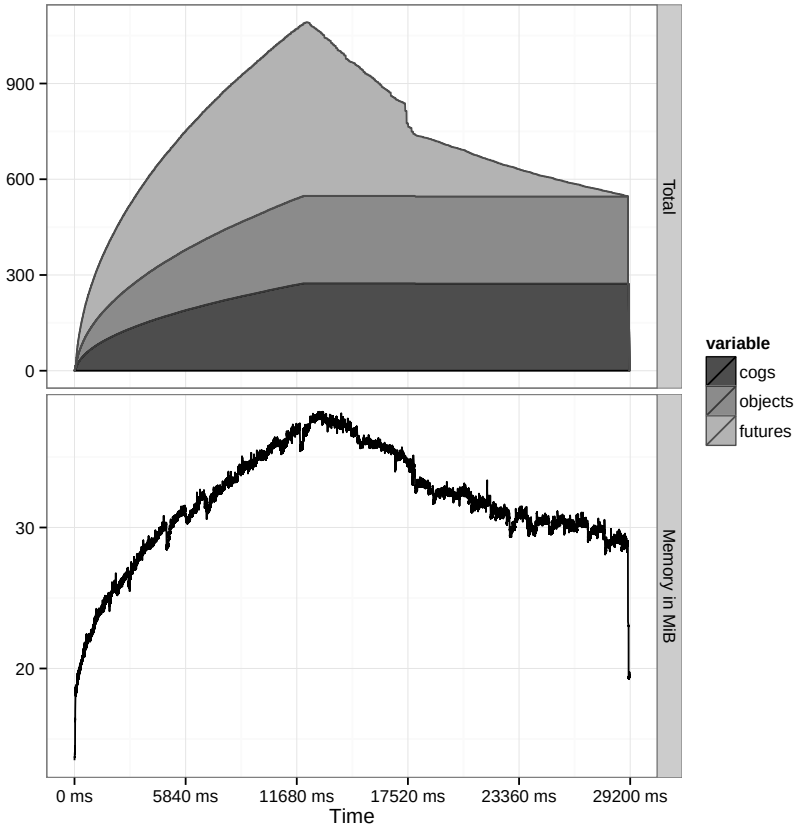

Table D.2: Indexing - Always Collect - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	31.6 MiB	31.6 MiB	38.2 MiB
COGs	0	235.2	273	274
Objects	0	234.8	273	274
Futures	0	105.2	104	272
Root futures	0	153.4	165	274
Processes	24	1143.2	1136	1665
Φ	0.9 ‰	43.6 ‰	43.3 ‰	63.5 ‰

Table D.3: Indexing - Always Collect - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.6	0	273
Objects kept	0	192.5	235.5	274
Futures swept	0	1.1	0	39
Futures kept	0	97.5	88.5	270

Table D.4: Indexing - Always Collect - Intervals

	Mean	Median	Max	Total
Stop world	53108 µs	60082 µs	93309 µs	25704089 µs
Mark	6240 µs	5446 µs	20882 µs	3020215 µs
Sweep	343 µs	291 µs	4408 µs	166081 µs
Collection cycle	59691 µs	70836 µs	108651 µs	28890385 µs
Collector only	6583 µs	5766 µs	21434 µs	3186296 µs
Mutator only	569 µs	487 µs	29888 µs	275609 µs

D.2.3 Trigger on Timeout

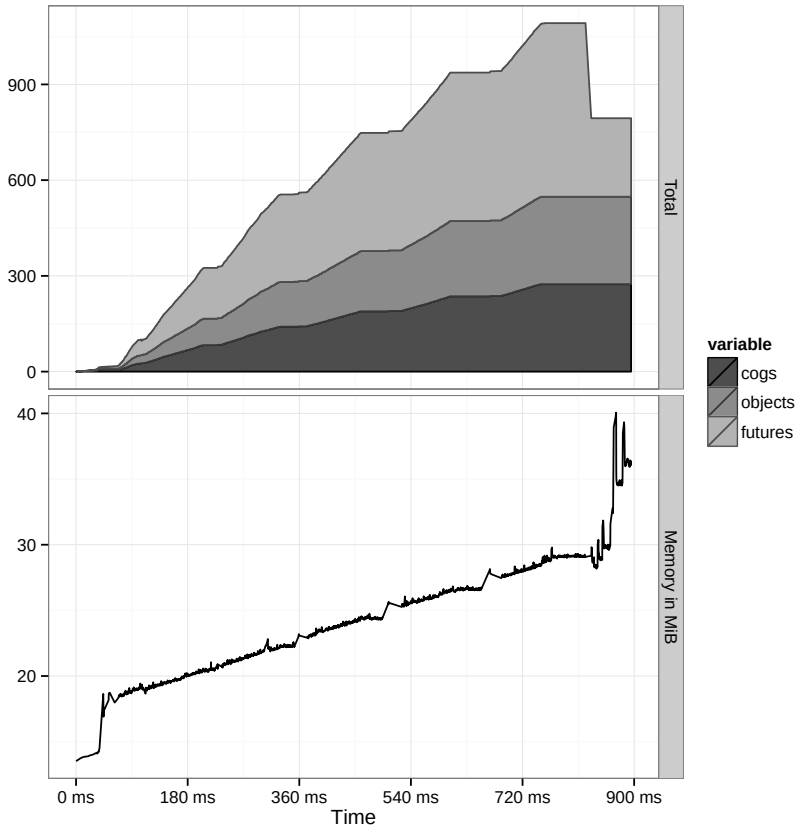

Table D.5: Indexing - Collect on time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	24.7 MiB	24.4 MiB	40.1 MiB
COGs	0	176.5	189	274
Objects	0	176.3	189	274
Futures	0	170.1	177.5	341
Root futures	0	153.2	174	273
Processes	24	1031.7	1150	1663
Φ	0.9 ‰	39.4 ‰	43.9 ‰	63.4 ‰

Table D.6: Indexing - Collect on time - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	0
Objects kept	26	158.2	165	274
Futures swept	0	50.5	0	298
Futures kept	20	115	106.5	231

Table D.7: Indexing - Collect on time - Intervals

	Mean	Median	Max	Total
Stop world	31440 µs	29996 µs	54606 µs	188637 µs
Mark	6217 µs	6658 µs	12093 µs	37302 µs
Sweep	431 µs	382 µs	935 µs	2584 µs
Collection cycle	38087 µs	37121 µs	63426 µs	228523 µs
Collector only	6648 µs	7125 µs	12686 µs	39886 µs
Mutator only	100335 µs	100290 µs	100793 µs	602009 µs

D.2.4 Trigger on Timeout with Stopping Running Tasks

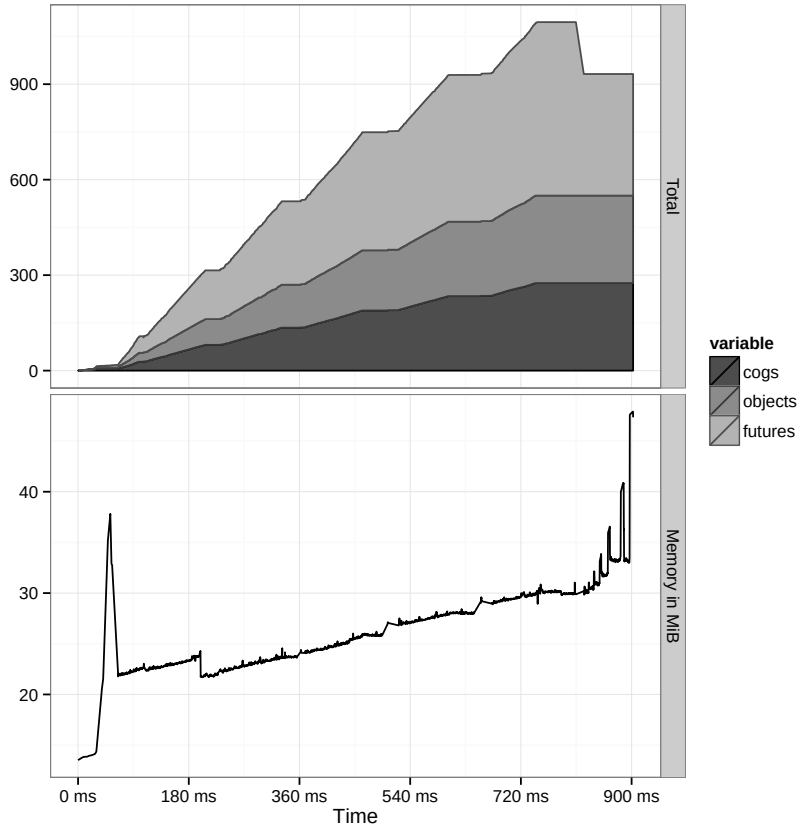

Table D.8: Indexing - Collect on time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	26.4 MiB	25.9 MiB	47.9 MiB
COGs	0	176.7	189	275
Objects	0	176.6	189	275
Futures	0	177.4	183	382
Root futures	0	155.1	175	275
Processes	24	1043.6	1151	1669
Φ	0.9 ‰	39.8 ‰	43.9 ‰	63.7 ‰

Table D.9: Indexing - Collect on time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	0
Objects kept	28	157	162	275
Futures swept	0	27.8	0	163
Futures kept	21	133	145.5	228

Table D.10: Indexing - Collect on time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	29784 µs	28876 µs	54647 µs	178701 µs
Mark	6545 µs	6538 µs	11793 µs	39270 µs
Sweep	393 µs	388 µs	654 µs	2356 µs
Collection cycle	36721 µs	35802 µs	67036 µs	220327 µs
Collector only	6938 µs	6926 µs	12389 µs	41626 µs
Mutator only	100274 µs	100267 µs	100464 µs	601644 µs

D.2.5 Trigger on Count

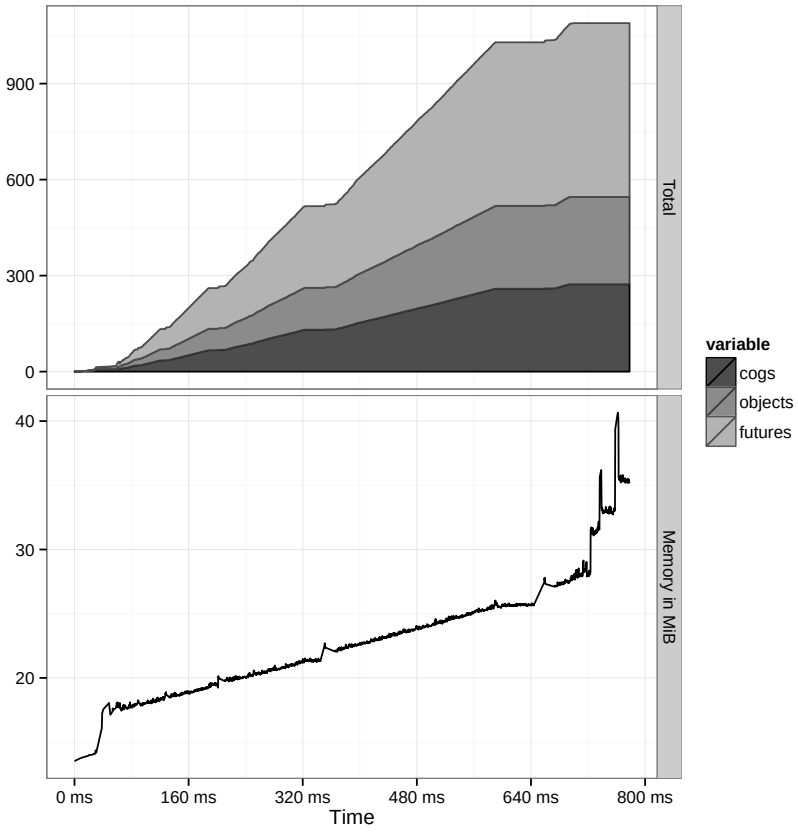

Table D.11: Indexing - Collect on count - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	23.3 MiB	22.9 MiB	40.7 MiB
COGs	0	162.3	163	273
Objects	0	162.1	163	273
Futures	0	175	158	543
Root futures	0	143.6	135	272
Processes	24	974.9	997	1658
Φ	0.9 ‰	37.2 ‰	38 ‰	63.2 ‰

Table D.12: Indexing - Collect on count - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0	0	0
Objects kept	8	75.7	35	259
Futures swept	0	0.9	0	6
Futures kept	3	70.7	30	254

Table D.13: Indexing - Collect on count - Intervals

	Mean	Median	Max	Total
Stop world	14376 µs	6106 µs	54589 µs	100632 µs
Mark	3298 µs	1161 µs	13353 µs	23084 µs
Sweep	175 µs	87 µs	572 µs	1227 µs
Collection cycle	17849 µs	7354 µs	68514 µs	124943 µs
Collector only	3473 µs	1248 µs	13925 µs	24311 µs
Mutator only	76180 µs	60382 µs	239159 µs	533257 µs

D.2.6 Trigger on Count or Timeout

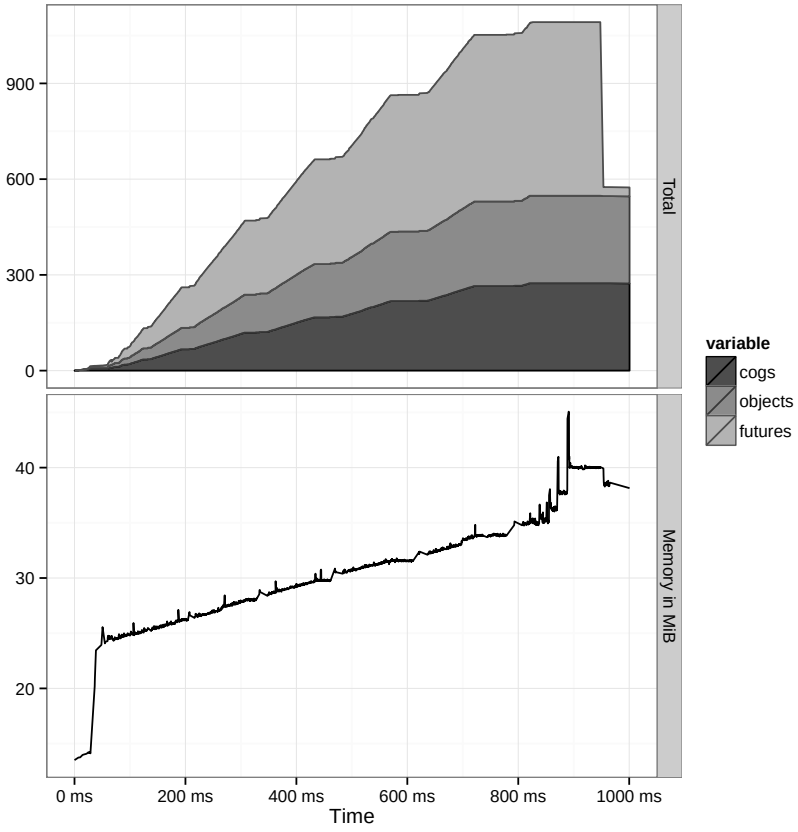

Table D.14: Indexing - Collect on count or time - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	31.2 MiB	31.2 MiB	45.1 MiB
COGs	0	178.5	198	274
Objects	0	178.4	198	274
Futures	0	204.9	186	516
Root futures	0	140.1	146	273
Processes	24	1046.7	1169	1664
Φ	0.9 ‰	39.9 ‰	44.6 ‰	63.5 ‰

Table D.15: Indexing - Collect on count or time - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.1	0	1
Objects kept	9	118.6	93.5	273
Futures swept	0	52.1	0	516
Futures kept	0	86.3	46	260

Table D.16: Indexing - Collect on count or time - Intervals

	Mean	Median	Max	Total
Stop world	23075 µs	15924 µs	58253 µs	230749 µs
Mark	4309 µs	3052 µs	12802 µs	43087 µs
Sweep	328 µs	227 µs	1146 µs	3277 µs
Collection cycle	27711 µs	19516 µs	71626 µs	277113 µs
Collector only	4636 µs	3592 µs	13373 µs	46364 µs
Mutator only	67601 µs	82530 µs	100424 µs	676012 µs

D.2.7 Trigger on Count or Timeout with Stopping Running Tasks

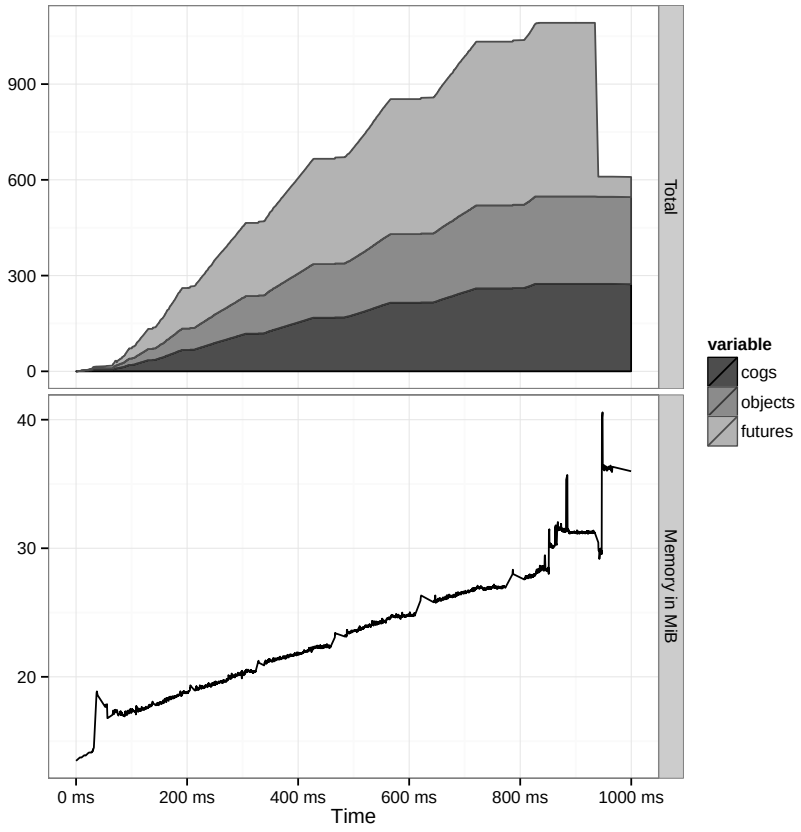

Table D.17: Indexing - Collect on count or time with stopping processes - Memory, counts and process ratio

	Min	Mean	Median	Max
Memory	13.5 MiB	24.1 MiB	24.1 MiB	40.6 MiB
COGs	0	177.6	197	274
Objects	0	177.5	197	274
Futures	0	195.3	179	481
Root futures	0	142.5	145	273
Processes	24	1038.9	1123	1663
Φ	0.9 ‰	39.6 ‰	42.8 ‰	63.4 ‰

Table D.18: Indexing - Collect on count or time with stopping processes - Sweeps

	Min	Mean	Median	Max
Objects swept	0	0.1	0	1
Objects kept	9	117.7	92.5	273
Futures swept	0	48.6	0	481
Futures kept	0	85.6	46	255

Table D.19: Indexing - Collect on count or time with stopping processes - Intervals

	Mean	Median	Max	Total
Stop world	21876 µs	14767 µs	52373 µs	218762 µs
Mark	4204 µs	3157 µs	11908 µs	42042 µs
Sweep	312 µs	220 µs	1085 µs	3116 µs
Collection cycle	26392 µs	18206 µs	64805 µs	263920 µs
Collector only	4516 µs	3439 µs	12432 µs	45158 µs
Mutator only	67628 µs	85272 µs	100953 µs	676285 µs

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0262011530.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Raleigh, NC, and Dallas, TX: The Pragmatic Programmers, LLC, 2007. ISBN: 978-1-9343560-0-5. URL: http://www.pragprog.com/titles/jaerlang/programming-erlang.

[3] Denis Caromel, Guillaume Chazarain, and Ludovic Henrio. "Garbage Collecting the Grid: A Complete DGC for Activities". English. In: *Middleware 2007*. Ed. by Renato Cerqueira and RoyH. Campbell. Vol. 4834. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 164–183. ISBN: 978-3-540-76777-0. DOI: 10.1007/978-3-540-76778-7_9. URL: http://dx.doi.org/10.1007/978-3-540-76778-7_9.

[4] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[5] Edsger W. Dijkstra et al. "On-the-fly Garbage Collection: An Exercise in Cooperation". In: *Commun. ACM* 21.11 (Nov. 1978), pp. 966–975. ISSN: 0001-0782. DOI: 10.1145/359642.359655. URL: http://doi.acm.org/10.1145/359642.359655.

[6] *Erlang Run-Time System Application (ERTS) Reference Manual*. Ericsson AB. Dec. 2014. URL: http://www.erlang.org/doc/apps/erts/index.html.

[7] J. F. Gimpel. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4". In: *Commun. ACM* 16.2 (Feb. 1973), pp. 91–100. ISSN: 0001-0782. DOI: 10.1145/361952.361960. URL: http://doi.acm.org/10.1145/361952.361960.

[8] Georg Göri. "ABS2Erlang". Draft of master's thesis to be submitted to the Graz University of Technology in 2015. The title is subject to change. 2014.

[9] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. “Deployment Variability in Delta-Oriented Models”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 304–319. ISBN: 978-3-662-45233-2. DOI: 10.1007/978-3-662-45234-9_22. URL: http://dx.doi.org/10.1007/978-3-662-45234-9_22.

[10] Einar Broch Johnsen et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. Ed. by Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 142–164.

[11] Richard Jones, Anthony Hosking, and Eliot Moss. *The Garbage Collection Handbook*. CRC Press, 2012. URL: http://www.pragprog.com/titles/jaerlang/programming-erlang.

[12] Niels Christian Juul. “Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System, Emerald”. PhD thesis. 1992.

[13] P. Lincoln M. Clavel S. Eker and J. Meseguer. “Principles of Maude”. In: *Electronic Notes in Theoretical Computer Science*. Ed. by J. Meseguer. Vol. 4. Elsevier Science Publishers, 2000.

[14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[15] *The ABS Language Specification*. June 2014. URL: http://tools.hats-project.eu/download/absrefmanual.pdf.

[16] Robert Virding. “A garbage collector for the concurrent real-time language Erlang”. English. In: *Memory Management*. Ed. by HenryG. Baler. Vol. 986. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 343–354. ISBN: 978-3-540-60368-9. DOI: 10.1007/3-540-60368-9_33. URL: http://dx.doi.org/10.1007/3-540-60368-9_33.